12

Stanford Artificial Intelligence Laboratory
Memo AIM-308

Computer Science Department
Report No. STAN-CS-77-641

November 1977

AUTOMATIC CONSTRUCTION OF ALGORITHMS AND DATA STRUCTURES USING A KNOWLEDGE BASE OF PROGRAMMING RULES

by

David R. Barstow

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*) 14

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 STAN-CS-77-641, AIM-308	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) 6 Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules.		5. TYPE OF REPORT & PERIOD COVERED 9 Technical rept.,
		6. PERFORMING ORG. REPORT NUMBER AIM-308
7. AUTHOR(s) 10 David R. Barstow		8. CONTRACT OR GRANT NUMBER(s) 15 MDA903-76-C-0206 ARPA Order-2494
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Arpa Order 2494
11. CONTROLLING OFFICE NAME AND ADDRESS Euguene Stubbs ARPA/PM 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE 11 Nov~~ember~~ 1977
		13. NUMBER OF PAGES 220 12 222p.
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) Philip Surra, ONR Representative Durand Aeronautics Building, Rm 165 Stanford University Stanford, CA 94305		15. SECURITY CLASS. ~~(of this report)~~ 15
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Releasable without limitation on dissemination

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

See back

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

094 120 JOB

Block 20.

Despite the wealth of programming knowledge available in the form of textbooks and articles, comparatively little effort has been applied to the codification of this knowledge into machine-usable form. The research reported here has involved the explication of certain kinds of programming knowledge to a sufficient level of detail that it can be used effectively by a machine in the task of constructing concrete implementations of abstract algorithms in the domain of symbolic programming.

Knowledge about several aspects of symbolic programming has been expressed as a collection of four hundred refinement rules. The rules deal primarily with collections and mappings and ways of manipulating such structures, including several enumeration, sorting and searching techniques. The principle representation techniques covered include the representation of sets as linked lists and arrays (both ordered and unordered), and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). In addition to these general constructs, many low-level programming details are covered (such as the use of variables to store values).

To test the correctness and utility of these rules, a computer system (called PECOS) has been designed and implemented. Algorithms are specified to PECOS in a high-level language for symbolic programming. By repeatedly applying rules from its knowledge base, PECOS gradually refines the abstract specification into a concrete implementation in the target language. When several rules are applicable in the same situation, a refinement sequence can be split. Thus, PECOS can actually construct a variety of different implementations for the same abstract algorithm.

PECOS has successfully implemented algorithms in several application domains,

including sorting and concept formation, as well as algorithms for solving the reachability problem in graph theory and for generating prime numbers. PECOS's ability to construct programs from such varied domains suggests both the generality of the rules in its knowledge base and the viability of the knowledge-based approach to automatic programming.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or any agency of the U. S. Government.

Stanford Artificial Intelligence Laboratory November 1977
Memo AIM-308

Computer Science Department
Report No. STAN-CS-77-641

AUTOMATIC CONSTRUCTION OF ALGORITHMS AND DATA STRUCTURES USING A KNOWLEDGE BASE OF PROGRAMMING RULES

by

David R. Barstow

ABSTRACT

Despite the wealth of programming knowledge available in the form of textbooks and articles, comparatively little effort has been applied to the codification of this knowledge into machine-usable form. The research reported here has involved the explication of certain kinds of programming knowledge to a sufficient level of detail that it can be used effectively by a machine in the task of constructing concrete implementations of abstract algorithms in the domain of symbolic programming.

Knowledge about several aspects of symbolic programming has been expressed as a collection of four hundred refinement rules. The rules deal primarily with collections and mappings and ways of manipulating such structures, including several enumeration, sorting and searching techniques. The principle representation techniques covered include the representation of sets as linked lists and arrays (both ordered and unordered), and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). In addition to these general constructs, many low-level programming details are covered (such as the use of variables to store values).

To test the correctness and utility of these rules, a computer system (called PECOS) has been designed and implemented. Algorithms are specified to PECOS in a high-level language for symbolic programming. By repeatedly applying rules from its knowledge base, PECOS gradually refines the abstract specification into a concrete implementation in the target language. When several rules are applicable in the same situation, a refinement sequence can be split. Thus, PECOS can actually construct a variety of different implementations for the same abstract algorithm.

PECOS has successfully implemented algorithms in several application domains, including sorting and concept formation, as well as algorithms for solving the reachability problem in graph theory and for generating prime numbers. PECOS's ability to construct programs from such varied domains suggests both the generality of the rules in its knowledge base and the viability of the knowledge-based approach to automatic programming.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or any agency of the U. S. Government.

ACKNOWLEDGEMENTS

My adviser, Cordell Green, has been a constant source of motivation and direction for this research. I especially appreciate his willingness to listen to half-baked ideas and his ability to focus on the central issues in our discussions. The other members of my reading committee, Bruce Buchanan, Don Knuth, and Terry Winograd, have all provided valued help and advice, not only while I was writing this thesis, but throughout my education at Stanford. I would also like to thank Randy Davis, Cathy De Young, Drew McDermott, Nils Nilsson, and Steve Wood for reading and discussing earlier drafts.

This research has benefited greatly from my interaction with the other members of the PSI project: Dick Gabriel, Jerry Ginsparg, Elaine Kant, Juan Ludlow, Brian McCune, Jorge Phillips, Lou Steinberg, and Ronny van den Heuvel. I would especially like to thank Brian for his help in designing the program model language, Juan for helping me to see the inadequacies of many of the rules before it was too late, and Elaine for the pleasure of working together on PSI's synthesis phase. I would also like to thank Bob Bolles, whose friendship made my life at Stanford considerably more enjoyable.

The Stanford A. I. Lab has provided the computer support and intellectual atmosphere which facilitated this work. ARPA and IBM have provided financial support.

Finally, I would like to thank my wife, Linda, for providing support and encouragement throughout this project, for doing far more than her share of keeping our house and home together, and for not letting me forget that there is more to life than is found in a computer lab.

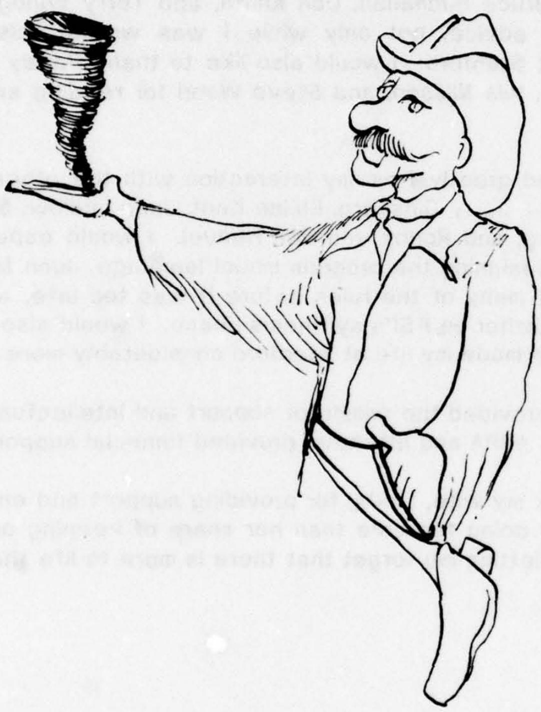


Table of Contents

Table of Contents

Table of Contents

Table of Contents

1. INTRODUCTION

Although large amounts of programming knowledge are available to human programmers in the form of books and articles, very little of this knowledge is available in a form suitable for use by a machine in performing programming tasks automatically. The principal goal of the research reported here is the explication of programming knowledge to a sufficient level of detail that it can be used effectively by a machine. The programming task considered in this experiment is that of constructing concrete implementations of abstract algorithms in the domain of symbolic programming. Knowledge about several aspects of symbolic programming has been expressed as a collection of four hundred refinement rules. The rules deal primarily with collections and mappings and ways of manipulating such structures, including several enumeration, sorting and searching techniques. The principal representation techniques covered include the representation of sets as linked lists and arrays (both ordered and unordered), and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). In addition to these general constructs, many low-level programming details are covered (such as the use of variables to store values).

To test the correctness and utility of these rules, a computer system (called PECOS) has been designed and implemented. Algorithms are specified to PECOS in a high-level language for symbolic programming. By repeatedly applying rules from its knowledge base, PECOS gradually refines the abstract specification into a concrete implementation in the target language. Currently, the target language is LISP (in particular, a subset of INTERLISP **[Teitelman 1975]**). Preliminary experiments indicate that PECOS can be fairly easily extended to deal with SAIL (an ALGOL-like language) **[Ludlow 1977]**. PECOS has successfully implemented algorithms in several application domains, including sorting and concept formation, as well as algorithms for solving the reachability problem in graph theory and for generating prime numbers.

Since the rules embody programming knowledge about several different techniques for implementing abstract constructs, PECOS can actually produce a variety of implementations for a single abstract algorithm. The primary value of such variability is that different implementations are appropriate under different circumstances. Efficiency considerations (such as expected set sizes or even the cost function itself) play a major role in the relative utility of different implementations. Constraints on the representation of the input and output also influence the suitability of a given implementation in a particular situation.

PECOS can be used under two different operational paradigms. In an interactive mode, when more than one rule is applicable, the user is allowed to select which should be applied (and, hence, which implementation will be constructed). For the convenience of the user, about a dozen choice-making heuristics have been added to PECOS. Experience indicates that these can handle about two-thirds of the choices that typically arise. If a user is uncertain about which rule is "best" for his or her purposes, PECOS can apply each in parallel, constructing a separate implementation for each rule that is applied.

PECOS also operates as the Coding Expert of the PSI program synthesis system [Green 1976]. In this role, choices between rules are made by an automated Efficiency Expert (known as LIBRA) that incorporates more sophisticated techniques than the simple heuristics mentioned above [Kant 1977]. The capability of developing different implementations in parallel is used extensively in the interaction between PECOS and LIBRA.

Although PECOS has been fairly successful, the long-term benefits of this research lie not in this particular implementation, but more in the rules themselves, for they help to formalize programming knowledge that has previously been available only informally. The variability of the domains in which the rules have been successfully applied indicates a fairly high degree of generality in the rules.

1.1. Guide to the reader

The rest of section 1 provides a general introduction to knowledge-based automatic programming. Section 2 is a concrete example of PECOS in operation. These sections set much of the stage for the rest of this thesis.

Sections 3, 4, and 5 provide more detailed discussions of PECOS's refinement paradigm, rule representation, and control structure. While such details are central to the operation of any rule-based system, these three sections can be skimmed without loss of continuity. In particular, later sections do not depend on learning the conventions of the rule representation.

Section 6 presents a detailed discussion of the heart of the PECOS experiment: a knowledge base of programming rules. Taken together, these rules constitute a detailed codification of knowledge about several different aspects of symbolic programming. While the entire rule set may exceed the casual reader's interest, it is hoped that individual subsections will be useful to those concerned about particular programming topics. Most of the rules are independent of the fact that PECOS's target language is LISP and can be understood without knowing LISP to any great detail. Programming experience in some language, however, certainly contributes to an understanding of the details included in the rules.

The next few sections are intended to characterize the rules and the kinds of programs they can successfully deal with. Section 7 presents a representative sample of the programs that PECOS can handle. Section 8 presents the results of several experiments designed to show how PECOS's capabilities changed as the knowledge base was increased. These sections are relatively important for understanding what part of programming has been codified and what part has not. Section 9 presents a short discussion of the nature of the refinement trees generated by PECOS's rules.

PECOS was developed as one of the modules of the PSI program synthesis system, and section 10 contains a discussion of the interaction between PECOS and PSI.

The last three sections summarize the results of this experiment. Section 11 is a retrospective discussion of my experiences in building a rule-based system for automatic programming. Hopefully these lessons will be of value to those readers interested in building their own rule-based systems for other tasks. Section 12 discusses some directions for further research suggested by this work. Finally, section 13 summarizes the conclusions that can be drawn from the PECOS experiment.

The appendix contains a complete list of all of the constructs available in PECOS's pattern matching facility.

1.2. Knowledge-based computer systems

The role of task-specific knowledge has become increasingly important in recent artificial intelligence work. MACSYMA, for example, embodies very large amounts of knowledge about mathematics and symbol manipulation **[Macsyma 1974]**. DENDRAL and MYCIN both depend on large collections of rules characterizing aspects of chemistry and infectious disease **[Buchanan and Lederberg 1971, Shortliffe 1974]**. AM uses several hundred specific heuristics to expand on its core of knowledge about elementary mathematics **[Lenat 1976]**. The central feature of all of these systems is that their performance is based not on their application of a few general principles, but on their access to large amounts of task-specific knowledge.

The basic methodology involved in developing such systems is to express knowledge about the system's task in a machine-usable form. One form that has been used with some success is the separation of the knowledge into relatively small, identifiable chunks[1]. One of the primary benefits of using such a form is the relative ease with which the knowledge base can be changed: new chunks can be added and old chunks changed[2]. Another benefit is the possibility of using the same knowledge for several different purposes (e.g., both forward- and backward-chaining). A third benefit comes from the potential for the system to explain its own actions so that human users need not take its conclusions on blind faith **[Davis 1976]**. Finally, the existence of identifiable chunks of knowledge about a particular domain can be of value to human experts in the domain by suggesting ways of stating and organizing knowledge that has often only been available informally. For example, geologists have expressed a great deal of interest in PROSPECTOR's knowledge base **[Duda et al 1977]**. Perhaps the central issue here is one of accessibility -- expressing domain-specific knowledge as small chunks makes it accessible for a variety of purposes.

----

[1] A separate issue concerns the way that such chunks are organized in the knowledge base.

[2] While this oversimplifies the situation somewhat, experience with such systems has been relatively successful in this regard (e.g., MYCIN **[Davis, Buchanan and Shortliffe 1977]**).

1.3. Knowledge-based automatic programming

The PECOS experiment has been an application of this knowledge-based approach to the development of an automatic programming system. The primary kind of knowledge involved is knowledge about the process of designing data structures and algorithms. The principal method used to identify this knowledge has been to examine particular algorithms, programs, and representation techniques and to try to identify the individual reasoning steps involved in their design. The knowledge involved in making such steps is then the knowledge to be codified into machine-useable form: each step is reflected by a single rule. A representative sample of these rules is presented below (in English, for the sake of clarity):

> ***A sequential collection may be represented as a linked list.***
>
> ***If a linked list is represented as a LISP list without a special header cell, then a retrieval of the first element in the list may be implemented as a call to the function CAR.***
>
> ***If a linked list is represented as a LISP list without a special header cell, then a test of whether an item is stored in an element cell of the list may be refined into a call to the LISP function MEMBER.***
>
> ***If the enumeration order is the same as the stored order of a collection, then the state of the enumeration may be saved as a location in the collection.***
>
> ***A collection may be represented as a mapping from items to Boolean values.***
>
> ***If an element X was determined by retrieving the element at location L of a sequential collection C, then L is the location of X in C.***

Note that the rules deal with specific, detailed aspects of symbolic programming. Note also that the rules are defined using specific programming concepts rather than in terms of goals or transformations on world models. The LISP function MEMBER is described explicitly as a way of testing whether an item is stored in an element cell of a linked list, rather than in terms of an output predicate defined over objects satisfying an input predicate. Another feature of the rules is that they explicitly mention decisions that are often only implicit in the final implementation. For example, "enumeration order" refers to the order in which elements of a stored collection are enumerated. An implicit part of a decision to trace down successive links in a list is a decision that the stored order of the list is the desired enumeration order.

Perhaps the greatest single benefit of the use of "small" rules is that the knowledge embedded in such rules can be applied in a variety of situations. As a simple example, consider the derivations of an enumerator over an array and an enumerator over a linked list. Many of the reasoning steps are shared by the two derivations. By breaking the derivation down into simple steps (as opposed to having two large rules, one for each derivation), the knowledge relevant to both derivations need not be repeated. This breakdown has been achieved primarily through the use of intermediate-level abstractions. For example, a "sequential collection" is more concrete than a collection and more abstract than a linked list or array. Most of the enumeration rules operate at the level of sequential collections.

Such a use of abstractions has recently been applied in several other areas. Verification systems, for example, have recently begun to incorporate the use of abstractions [von Henke and Luckham 1974, Robinson and Levitt 1977]. Abstraction is a key element in such languages as ALPHARD and CLU [Wulf, London and Shaw 1976, Liskov et al 1977]. The major reason for the use of abstractions is that they help to reduce the complexity of large programs. Abstractions enable systems (or people) to concentrate on important aspects while ignoring minor details. The use of abstractions in PECOS shares this motivation.

1.4. Program construction through gradual refinement

As noted earlier, through the successive application of its rules, PECOS gradually refines the original abstract specification into a concrete LISP implementation. The process may be viewed as the construction of a sequence of program descriptions. The first description in the sequence represents the abstract specification and the final description represents the concrete implementation. Each rule application produces the next description by adding a small amount of detail. While constructing such a sequence, there will be many situations in which more than one rule is applicable. Under such circumstances, PECOS can apply each rule separately, causing the refinement sequence to split into several sequences. Thus, PECOS can actually construct a refinement tree in which each path from the root to a leaf is a refinement sequence. Each leaf represents a different concrete implementation of the abstract algorithm represented by the root[3].

Similar notions of refinement have recently gained importance in the area of programming methodology. "Structured programming" and "stepwise refinement", for example, are programming techniques based on the gradual refinement of program statements until constructs available in the target language have been reached [Dahl, Dijkstra and Hoare 1972, Wirth 1971]. Despite its success in human programming efforts, relatively little work has been applied to its use in automatic programming systems.

----
[3] This space of alternative implementations is precisely the space explored by PSI's efficiency expert.

1.5. Approaches to automatic programming

The term "automatic programming" has a long history. It was used as early as 1954 to refer to the development of programming languages **[MCAP 1954]**. More generally, the term is used to describe attempts to automate various parts of the programming process. Several approaches have been (and continue to be) used in solving the problem.

1.5.1. Codification of programming knowledge

Although large amounts of programming knowledge are available to people in the form of books and articles, comparatively little work has been done on the codification of this knowledge into machine-usable form. The one notable exception is the identification and collection of optimizing transformations. Standish, for example, has a collection of several hundred **[Standish et al 1976]**. Low's system, with its knowledge of seven different representations, is one form of codified knowledge about sets **[Low 1974]**. In related work, Rovner has identified several techniques for representing associative triples **[Rovner 1976]**. Ruth codified some of the aspects of simple sorting for the purpose of automating the analysis of student programs **[Ruth 1976]**. The work presented here is, in part, a continuation of previous attempts to codify knowledge about sorting **[Green and Barstow 1975, 1977a, 1977b]**.

In their work on the development of a program analysis system as part of a programmer's apprentice, Rich and Shrobe have codified some of the knowledge involved in hash table programs, but their work has concentrated on representational issues **[Rich and Shrobe 1976]**. They discuss similar notions of refinement and knowledge base organization, although few details are given.

1.5.2. High-level languages

Another trend in programming methodology has been the development of high-level languages incorporating increasingly abstract constructs. SETL, for example, includes various set operations **[Schwartz 1975]**. As such constructs have become further abstracted from constructs available at the machine level, different techniques for determining data structure representations and operation implementations have been developed. Low's system, for example, used a partitioning and hill-climbing technique to select the representation likely to be the most efficient in a particular situation **[Low 1974]**. The most significant point for comparison between Low's system and PECOS revolves around PECOS's use of intermediate-level abstractions. While Low's system refines an abstract data structure into a particular machine representation in a single step, PECOS's rules might involve four or five steps, each corresponding to a separate abstraction level. This use of abstractions helps to avoid one of the restrictions that Low's system was forced to make: under certain circumstances the arguments to various set operations were forced to have the same representation so that the number of conversion tables could be kept manageably small. In effect, the intermediate level abstractions enable PECOS to write the table entries as they are needed, rather than to keep them all stored.

1.5.3. Problem-solving and theorem-proving approaches

Much work in automatic programming has involved the use of general purpose theorem-provers and problem-solvers. Programs are usually specified in terms of input-output relations. For theorem-provers these are normally in the form of predicates over the input and output variables. Problem-solvers normally accept specifications in terms of initial and final (or goal) states expressed as assertions. In both cases, the primitive operations available are described in similar terms. The earliest work in problem-solving involved determining a single sequence of operations satisfying the input-output relation **[Green 1969, Fikes and Nilsson 1971]**. This has led to various ways of planning through the use of intermediate subgoals (e.g., NOAH **[Sacerdoti 1975]**) or programs that "almost" work (e.g., HACKER **[Sussman 1975]**). As these systems have become more sophisticated, the general progression has been away from general purpose inference systems and toward systems designed specifically for program manipulation **[Manna and Waldinger 1977, Darlington and Burstall 1976]**. Despite this trend, most of this work has been aimed at identifying general programming principles that are relatively domain-independent.

In a sense, PECOS and such problem-solving systems are aimed at different tasks. PECOS assumes that the basic algorithm has already been determined, while the problem-solving approach is aimed at determining an algorithm when such an algorithm is not known. One could, in fact, imagine using such a problem-solver as a front end for an implementation system like PECOS: the target language of the problem-solver would be the specification language of the implementation system.

1.5.4. Program specification

A more central issue is that of program specification: what are the best ways for human users to specify programs for an automatic programming system to write? Experience suggests that different specification methods (e.g., input-output specifications, high-level languages, examples and traces, natural language, dialogue) are appropriate for different domains and even for different users **[Green et al 1974]**. Various research projects are developing techniques for handling such specifications. PSI's acquisition phase (see section 10) is aimed at allowing either dialogue or traces. The SAFE system is aimed at using informal English specifications **[Balzer, Goldman and Wile 1977]**.

2. A DETAILED EXAMPLE

In this section the use of programming rules to construct a particular program will be illustrated. In order to focus on the nature of the rules and the refinement process, the example will be presented entirely in English. The details of PECOS's internal representations will be covered in sections 3, 4 and 5.

After a description of the abstract algorithm to be implemented, several specific aspects will be discussed in detail. For each of these aspects, the abstract description of that part of the algorithm specification will be presented. Then a sequence of rules will be given, together with the refinements they produce in the original description. The result of this sequence of rule applications will be a concrete LISP implementation of the original abstract description.

One principal characteristic of these refinement sequences is the fairly small step size: each step produces a description that is only slightly more specific than the previous description. This is characteristic of the rules as well: each embodies a rather small, detailed "piece" of programming knowledge. As suggested in the introduction, one effect of this small rule size is that the knowledge embedded in the rule can be applied in a variety of situations. This will be seen in the examples in this section, as the same rules will be applied in several different situations.

2.1. The Reachability Problem

The example is based on a variant of the Reachability Problem [Thorelli 1972]:

> Given a directed graph, *G*, and an initial vertex, *v*, find the vertices reachable from *v* by following zero or more arcs.

The problem can be solved with the following algorithm:

> Mark *v* as a boundary vertex and mark the rest of the vertices of *G* as unexplored. If there are any vertices marked as boundary vertices, select one, mark it as explored, and mark each of its unexplored successors as a boundary vertex. Repeat until there are no more boundary vertices. The set of vertices marked as explored is the desired set of reachable vertices.

Note that the algorithm's major actions involve manipulating a mapping of vertices to markings. Based on this observation, the algorithm can be expressed at the level of PECOS's specification language. The following is an English paraphrase of the specification given to PECOS when this example was run. (As a notational convenience, X[Y] will be used to denote the image of Y under the mapping X and X^{-1}[Z] will be used to denote the inverse image of Z under X. Note that the inverse image is a collection of domain elements, while the image is a single range element.)

```
DATA STRUCTURES
        VERTICES        a collection of integers
        SUCCESSORS      a mapping of integers to collections of integers
        START           an integer
        MARKS           a mapping of integers to
                                  {"EXPLORED", "BOUNDARY", "UNEXPLORED"}

ALGORITHM
        VERTICES ← input a list of integers;
        SUCCESSORS ← input an association list of <integer, list of integers> pairs;
        START ← input an integer;
        for all X in VERTICES:
                MARKS[X] ← "UNEXPLORED";
        MARKS[START] ← "BOUNDARY";
        repeat until MARKS⁻¹["BOUNDARY"] is empty:
                X ← any element of MARKS⁻¹["BOUNDARY"];
                MARKS[X] ← "EXPLORED";
                for all Y in SUCCESSORS[X]:
                        if MARKS[Y] = "UNEXPLORED" then MARKS[Y] ← "BOUNDARY";
        output MARKS⁻¹["EXPLORED"] as a list of integers.
```

The specification is abstract enough that several significantly different implementations are possible. For example, MARKS could be represented as an association list of <integer, mark> pairs or as an array whose entries are the marks. The relative efficiency of these implementations varies considerably with several factors. For example, if the set of vertices (integers) is relatively sparse in a large range of possible values, then implementing MARKS as an array with a separate index for each possible value would probably require too much space, and an association list would be preferable. On the other hand, if the set of vertices is dense or the range small, an array might allow much faster algorithms because of the random-access capabilities of arrays. For the remainder of this discussion, it will be assumed that the range of possible values for the vertices is small enough that array representations are feasible. (When the example was run, a range of 1 to 100 was specified.) Note also that concrete input representations are specified for VERTICES (a linked list), SUCCESSORS (an association list), and START (an integer), and that an output representation is specified for MARKS^{-1}["EXPLORED"] (a linked list). These constrain the input and output but not the internal representation. They are intended to reflect the desires of some hypothetical user and PECOS could handle other input and output representations equally well.

When PECOS was run on the Reachability Algorithm, there were several dozen situations in which more than one rule was applicable. In most of these cases, selecting different rules would result in the successful construction of different implementations. As mentioned in the introduction, PECOS has a set of about a dozen heuristics for selecting one rule over another. These heuristics were sufficient to select a rule in about two-thirds of the choice points. In the remaining third, a rule was selected interactively in order to construct the particular implementation.

2.2. SUCCESSORS

One of the major data structures in the Reachability Algorithm is the SUCCESSORS mapping. Under this mapping, the image of a vertex is the set of immediate successors of the vertex:

$$\text{SUCCESSORS}[v] = \{ x \mid v \to x \text{ in } G\}$$

SUCCESSORS is constrained to be an association list when it is input, but such a representation may require significant amounts of searching to compute SUCCESSORS[X]. Since this would be done in the inner loop, a significantly faster algorithm can be achieved by using an array representation with the entry at index *k* being the set of successors of vertex *k*. In the rest of this section, the derivation of this array representation will be considered in detail.

2.2.1. Representation of SUCCESSORS

SUCCESSORS is a mapping of integers to collections of integers. This abstract description may be summarized as shown below (an English paraphrase of PECOS's internal representation):

```
SUCCESSORS:
    MAPPING (integers → collections of integers)
```

The first representation decision for many abstract data structures is whether to represent the structure explicitly or implicitly. An explicit representation for a mapping involves indicating every <domain, range> pair explicitly. An implicit representation is one in which, for example, the image of a domain element is computed by some function. In this case, we will represent SUCCESSORS explicitly. So we apply the following rule (again, an English paraphrase of PECOS's internal representation):

A mapping may be represented explicitly.

The result of applying this rule is shown below:

```
SUCCESSORS:
    EXPLICIT MAPPING (integers → collections of integers)
```

The second decision is whether to store the pairs in a single structure or to keep them distributed in several structures (e.g., property list markings). Here we will use a single structure, applying the following rule:

An explicit mapping may be stored in a single structure.

with the following result:

SUCCESSORS:
STORED MAPPING (integers → collections of integers)

The next step involves selecting the type of structure to be used. There are many possibilities here, including tabular structures, discrimination nets, and sets of <domain, range> pairs. Applying the following rule:

A stored mapping with typical domain element X and typical range element Y may be represented with an association table whose typical key is X and whose typical value is Y.

gives us a tabular representation for SUCCESSORS in the inner loop[4]. The following description results[5]:

$\text{SUCCESSORS}_{\text{table}}$:
ASSOCIATION TABLE (integers → collections of integers)

The possibilities for tabular representations are dependent on the keys of the table. In the case of $\text{SUCCESSORS}_{\text{table}}$, each key is an integer from a fixed range, so an array representation can be used. The following rule is applied:

An association table whose typical key is an integer from a fixed range and whose typical value is Y may be represented as an array with typical entry Y.

and the following description results:

$\text{SUCCESSORS}_{\text{array}}$:
ARRAY (collection of integers)

----

[4] This is one of the situations in which a rule was chosen interactively. The other applicable rule is: ***A stored mapping with typical domain element X and typical range element Y may be represented as a stored collection whose typical element is a pair with DOMAIN part X and RANGE part Y.*** Had this other rule been applied, one could have derived, for example, an association list representation.

[5] Subscripts (as in $\text{SUCCESSORS}_{\text{table}}$) will be used to distinguish between representations at different refinement levels.

The final step in the representation involves the selection of a particular data structure in the target language. The following rule allows us to use the array representation available in the LISP dialect being used (INTERLISP):

> ***An array may be represented directly as a LISP array.***

Thus, through the application of five rules, SUCCESSORS has been refined from the abstract notion of a mapping into a particular concrete LISP representation:

```
SUCCESSORS_lisp:
    LISP ARRAY (collection of integers)
```

The next step is the representation of the objects to be stored in the array. Through a sequence of about six rule applications, a LISP LIST representation is developed. The sequence of rules is similar to that of the BOUNDARY set (see section 2.4) and will be omitted here. Their result is the final description of SUCCESSORS:

```
SUCCESSORS_lisp:
    LISP ARRAY (LISP LIST (integer))
```

2.2.2. SUCCESSORS[X]

Determining the set of successor vertices for a given vertex involves computing the image of that vertex under the SUCCESSORS mapping. The abstract specification of this operation is:

```
compute the image of X under SUCCESSORS
```

The construction of the program for computing SUCCESSORS[X] follows a line parallel to the determination of the representation of SUCCESSORS. The first rule is dependent on the fact that SUCCESSORS[X] is represented as an association table:

> ***If a mapping is stored as an association table, the image of a domain element X may be computed by retrieving the table entry associated with the key X.***

Applying this rule produces the following description:

```
retrieve the entry in SUCCESSORS_table for the key X
```

Similarly, the next rule depends on the representation of $\text{SUCCESSORS}_{\text{table}}$ as an array:

> ***If an association table is represented by an array, the entry for a key X may be retrieved by retrieving the array entry whose index is X.***

When this rule is applied, the following description results:

```
retrieve the entry in SUCCESSORS_array for the index X
```

Finally, a LISP-specific rule is applied:

> ***If an array is represented as a LISP array, the entry for an index X may be retrieved by applying the function ELT.***

yielding the LISP code for this part of the program:

```
(ELT SUCCESSORS_lisp X)
```

2.2.3. Converting between Representations of SUCCESSORS

Recall that the input representation for SUCCESSORS is constrained to be an association list of <integer, list of integers> pairs. The description corresponding to this representation is the following:

```
SUCCESSORS_input:
    LISP LIST (CONS CELL (DOMAIN . RANGE))
        DOMAIN: integer
        RANGE: LISP LIST (integer)
```

Since the input and internal representations differ, a representation conversion must be performed. This occurs when the association list representation is input. The original program description for the input operation is the following:

```
SUCCESSORS ← input a mapping (as an association list);
```

The following rule introduces the representation conversion:

If a mapping is input, its representation may be converted into any other representation before further processing.

When this rule is applied, the following description is produced:

```
SUCCESSORS_input ← input a mapping (as an association list);
SUCCESSORS ← Convert SUCCESSORS_input
```

The construction of a program for performing a conversion is generally dependent on both the initial and the final representations. In the case of the SUCCESSORS mapping, the first rule shows this dependence on the initial representation:

If a mapping is represented as a stored collection of pairs, it may be converted by considering all pairs in the collection and setting the image (under the new mapping) of the domain field of the pair to be the range field.

When the rule is applied to the convert operation, we have the following description:

```
For all X in SUCCESSORS_input:
    set SUCCESSORS[X:DOMAIN] to X:RANGE
```

where X:DOMAIN and X:RANGE signify the retrieval of the DOMAIN and RANGE parts of the pairs.

Since the pairs in SUCCESSORS$_{input}$ are represented as CONS cells, the X:DOMAIN an.' X:RANGE operations may be implemented easily through the application of one rule in each case.

If a pair is represented as a CONS cell and field X is stored in the CAR part of the cell, the value of field X may be retrieved by applying the function CAR.

```
(CAR X)
```

If a pair is represented as a CONS cell and field X is stored in

> ***the CDR part of the cell, the value of field X may be retrieved by applying the function CDR.***

```
(CDR X)
```

When these pieces of code are substituted into the previous description we have the following:

```
For all X in SUCCESSORSinput:
    set SUCCESSORS[(CAR X)] to (CDR X)
```

The implementation of the "set SUCCESSORS[(CAR X)]" operation is constructed by applying a sequence of rules similar to those used for implementing SUCCESSORS[X] in the previous section. The result of applying these rules is the following LISP code:

```
(SETA SUCCESSORSlisp (CAR X) (CDR X))
```

Substituting this into the "For all" construct, we have the following:

```
For all X in SUCCESSORSinput:
    (SETA SUCCESSORSlisp (CAR X) (CDR X))
```

We can now consider the derivation of the program for the "For all" construct. The first rule to be applied is the following:

> ***An operation of performing some action for all elements of a stored collection may be implemented by a total enumeration of the elements, applying the action to each element as it is enumerated.***

This rule effectively states that the action will be performed to one element at a time (as opposed to some kind of parallel control structure). It results in the following description:

```
Enumerate X in SUCCESSORSinput:
    (SETA SUCCESSORSlisp (CAR X) (CDR X))
```

The development of a structure for enumerating the elements of a stored collection involves several considerations. The first decision is the determination of the order in which the elements are to be enumerated. In many applications (such as sorting), this order may be constrained to be relative to some particular ordering relation. In this case, however, there is no such constraint, and the following rule may be applied:

> ***If the enumeration order is unconstrained, the elements of a stored collection may be enumerated in the order in which they are stored[6].***

The next consideration involves selecting some scheme by which the state of the enumeration can be saved on each iteration. The following rule can be applied here:

> ***If a stored collection is represented as a linked list and the enumeration order is the stored order, the state of the enumeration may be saved as a pointer to the list cell of the next element[7].***

The derivation path now proceeds through several steps based on the particular state-saving scheme chosen, including the determination of the initial state (a pointer to the first cell), a termination test (the LISP function NULL), and an incrementation step (the LISP function CDR). The end result is a loop approximated by the following description:

```
STATE ← $\text{SUCCESSORS}_{input}$;
loop:
          if (NULL STATE) then exit;
          X ← (CAR STATE) ;
          (SETA $\text{SUCCESSORS}_{lisp}$ (CAR X) (CDR X)) ;
          STATE ← (CDR STATE) ;
          repeat;
```

The complete LISP code for this part is included in the listing of the final Reachability Program in section 2.6.

----

[6] This is actually a slight simplification; the complete rule also reflects a dependence on viewing the collection as a "sequential collection". Sequential collections will be introduced in the discussion of the BOUNDARY set in section 2.4.

[7] This is also a simplification of a more general rule for sequential collections.

2.3. MARKS

MARKS is the principal data structure involved in the Reachability Algorithm. At each iteration through the loop it represents what is currently known about the reachability of each of the vertices in the graph:

MARKS[X] = "EXPLORED"
⇒ X is reachable and its successors have been noted as reachable
MARKS[X] = "BOUNDARY"
⇒ X is reachable and its successors have not been examined
MARKS[X] = "UNEXPLORED"
⇒ no path to X has yet been found

In the rest of this section, E, B, and U will denote "EXPLORED", "BOUNDARY", and "UNEXPLORED" respectively.

Note that the computation of the inverse image of some range element is a common operation on MARKS. In such situations, it is often convenient to use an inverted representation. That is, rather than associating range elements with domain elements, sets of domain elements can be associated with range elements. In this section, we will consider the derivation of such a representation for MARKS.

2.3.1. Representation of MARKS

MARKS is a mapping of integers to a collection of three elements, E, B, and U. The abstract description for MARKS is as follows:

```
MARKS:
    MAPPING (integers → {E,B,U})
```

The "inverted" option for mappings is available through the use of the following rule:

A mapping with typical domain element X and typical range element Y may be represented as a mapping with typical domain element Y and typical range element a collection with typical element X.

Applying this rule gives the following description:

```
MARKSinv:
    MAPPING ({E,B,U} → collections of integers)
```

At this point, the same two rules that were applied to SUCCESSORS can be applied to $\text{MARKS}_{\text{inv}}$:

A mapping may be represented explicitly.

```
MARKSinv:
    EXPLICIT MAPPING ({E,B,U} → collections of integers)
```

An explicit mapping may be stored in a single structure.

```
MARKSinv:
    STORED MAPPING ({E,B,U} → collections of integers)
```

Again we are faced with the selection of the structure in which the mapping is stored. In this case, we may take advantage of the fact that the domain is a fixed set of known alternatives (E, B, and U) and apply the following rule:

A stored mapping whose domain is a fixed set of alternatives and whose typical range element is Y may be represented as a plex with one field for each alternative and with each field being Y.

A plex is an abstract kind of record structure, consisting of a fixed set of named fields, each with an associated substructure, but without any particular commitment to the way the fields are stored in the plex. The description of $\text{MARKS}_{\text{plex}}$ is then as follows:

```
MARKSplex:
    PLEX (UNEXPLORED, BOUNDARY, EXPLORED)
        EXPLORED: collection of integers
        BOUNDARY: collection of integers
        UNEXPLORED: collection of integers
```

Of course, the rules for manipulating mappings represented in this way must insure that the three collections are mutually disjoint.

In LISP, the obvious way to represent such a structure is with CONS cells. The application of several rules dealing with such cells yields the following S-expression representation for $\text{MARKS}_{\text{plex}}$:

```
MARKS_lisp:
    CONS CELLS (UNEXPLORED BOUNDARY . EXPLORED)
        EXPLORED: collection of integers
        BOUNDARY: collection of integers
        UNEXPLORED: collection of integers
```

Notice that we are now concerned with three separate collections which need not be represented the same way. In fact, since they are used for different purposes, it may well be advantageous to represent them differently. The representations of BOUNDARY and UNEXPLORED will be considered in sections 2.4 and 2.5. First, however, we will look at some of the operations applied to MARKS.

2.3.2. $MARKS^{-1}$["BOUNDARY"]

The first operation we will consider is the computation of the inverse image of *B* under the MARKS mapping. The abstract description of this operation is as follows:

```
compute the inverse image of B under MARKS
```

With most representations for mappings, the computation of an inverse image can be relatively complex, possibly including an enumeration of all domain elements. In the case of MARKS, however, the mapping was inverted and the computation of the inverse is quite simple[8]. The following rule allows us to take advantage of this property:

If a mapping is represented as an inverted mapping, the inverse image of a range element X may be computed by computing the image of X under the inverted correspondence.

Applying this rule yields the following description:

```
compute the image of B under MARKS_inv
```

The next refinement step for $MARKS_{inv}$ was the decision to use a plex to represent the mapping. This is particularly useful if the domain element is known when the program is being constructed, as is the case here. (The domain element is *B*.) The following rule can then be applied:

----

[8] This is, presumably, the reason for inverting the mapping.

If a mapping is represented as a plex, the image of a known domain element X may be computed by retrieving the X field of the plex.

This yields the following description:

```
retrieve the BOUNDARY field of MARKSplex
```

The next two steps involve retrieving the field from the CONS cells used to represent MARKS_{plex}. The result is the following LISP code:

```
(CAR (CDR MARKSlisp))
```

2.3.3. Change MARKS[X] from "BOUNDARY" to "EXPLORED"

One of the operations applied frequently to the MARKS mapping is to change the image of a particular element. For example, after X (an element of MARKS^{-1}["BOUNDARY"]) has been chosen, one of the operations applied to X is the following:

```
change MARKS[X] from B to E
```

The refinement rule of this operation is dependent on the representation of MARKS as an inverted mapping.

If a mapping is represented as an inverted mapping, the operation of changing the image of a domain element X from Y to Z may be implemented by removing X from the image of Y and adding X to the image of Z under the inverted mapping.

When this rule is applied the following description results:

```
remove X from MARKSinv[B];
add X to MARKSinv[E]
```

The removal operation will be considered in more detail after determining a representation for the BOUNDARY collection.

2.4. BOUNDARY

BOUNDARY is the set of all vertices that map to *B* under MARKS. Since MARKS is inverted, this collection exists explicitly and a representation for it must be selected. The operations that are applied to BOUNDARY include the addition and deletion of elements and the selection of some element from the collection. For such operations, a linked list structure is often convenient. In this section we will consider the derivation of such a representation.

2.4.1. Representation of BOUNDARY

PECOS's internal representation for the description of BOUNDARY may be paraphrased as follows:

```
BOUNDARY:
    COLLECTION (integer)
```

As with mappings, the first decision is whether to use an explicit or an implicit representation. In the case of collections, an explicit representation is one in which each element is indicated explicitly. An implicit representation is one in which not all elements have such an explicit indication. For example, upper and lower bounds on a set of integers is an implicit representation. In the case of BOUNDARY, the following rule will be applied:

A collection may be represented explicitly.

yielding the following description:

```
BOUNDARY:
    EXPLICIT COLLECTION (integer)
```

Here again, the next decision is whether to keep all of the elements in a single structure or to use some kind of distributed representation. Applying the following rule:

An explicit collection may be stored in a single structure.

indicates a decision to store all of the elements together:

```
BOUNDARY:
    STORED COLLECTION (integer)
```

The next step is the selection of a particular structure. One common type of structure involves the use of some sequential arrangement of locations. Each location contains one element in the collection. (Another common type is to a tree structure.) The following rule can be applied:

> ***A stored collection with typical element X may be represented as a sequential arrangement of locations in which instances of X are stored.***

producing the following description:

```
BOUNDARYseq:
    SEQUENTIAL COLLECTION (integer)
```

Note that there is no commitment to any particular way of achieving the sequential arrangement. Both arrays and linked lists are reasonable alternatives here. Applying the following rule:

> ***A sequential arrangement of locations may be represented as a linked list.***

commits us to the use of some kind of linked list:

```
BOUNDARYlist:
    LINKED LIST (integer)
```

Again, there are still several possibilities. Under some circumstances, parallel arrays may be used, with one array containing the elements and one containing the links. The following rule takes the alternative of using cells allocated from free storage:

> ***A linked list may be represented using linked free cells.***

The resulting description is as follows:

```
BOUNDARYcells:
    LINKED FREE CELLS (integer)
```

It is often convenient to use a special header cell with such lists, so that the empty list need not be considered as a special case. Applying the following rule:

> ***A special header cell may be used with linked free cells.***

enables us to make use of this technique:

```
BOUNDARY_cells:
    LINKED FREE CELLS (integer) with special header cell
```

Any use of cells allocated from free storage requires allocation and garbage collection mechanisms. In LISP, both are available with the use of CONS cells, so we can apply the following rule:

> ***Linked free cells may be represented using a LISP list of CONS cells.***

Thus, the concrete data structure selected for representing the BOUNDARY collection is the following:

```
BOUNDARY_lisp:
    LISP LIST (integer) with special header cell
```

2.4.2. Any Element of MARKS^{-1}["BOUNDARY"]

The main loop of the algorithm is repeated until MARKS^{-1}["BOUNDARY"] (i.e., the BOUNDARY collection) is empty. The action at each iteration involves selecting some element from this collection. PECOS's representation of this operation may be paraphrased as:

```
    retrieve any element of BOUNDARY
```

Recall that one of the intermediate steps in the BOUNDARY derivation involved the use of a sequential collection. The first refinement step for the "any" operation is dependent on that step having been made. The relevant rule is:

> ***If a collection is represented as a sequential collection, the retrieval of any element in the collection may be implemented as the retrieval of the element at any location in the collection.***

Applying this rule yields:

```
    retrieve the element at location L of BOUNDARY_seq
        L is any location
```

The next step is then to select the location to be used. There are several possibilities for sequential collections, of which the two most useful are the front and the back. Of these, the front is generally best for linked lists; although the back can also be used, it is usually less efficient[9]. The decision to use the front can be taken by applying the following rule:

> ***If a location in a sequential collection is unconstrained, the front may be used.***

which produces the following description:

retrieve the element at the front of $\text{BOUNDARY}_{\text{list}}$

The next step is dependent on the representation of BOUNDARY as linked free cells with a special header cell. The appropriate rule here is:

> ***If a linked list is represented using linked free cells with a special header cell, the front location may be computed by retrieving the link from the first cell.***

When this rule is applied, we have:

retrieve the element from the cell indicated by the link from the first cell of $\text{BOUNDARY}_{\text{cells}}$

The computation of the link can be implemented by applying the following LISP-specific rule:

> ***If linked free cells are implemented as a LISP list, the link from the first cell may be computed by using the function CDR.***

Once the cell has been determined, the computation of the element can be implemented by applying the following rule:

> ***If linked free cells are implemented as a LISP list, the element at a cell may be computed by using the function CAR.***

The result of these two rule applications, when combined with the code for computing $\text{MARKS}_{\text{inv}}[B]$, is the following LISP code for computing "any element of MARKS^{-1}["BOUNDARY"]":

----

[9] The selection of the "front" rule over the "back" rule for linked lists is one of the choices made by PECOS's heuristics.

```
(CAR (CDR (CAR (CDR MARKSlisp))))
```

2.4.3. Remove X from $\text{MARKS}_{\text{inv}}$["BOUNDARY"]

Recall that one of the operations involved in changing the image of X from B to E is the removal of X from $\text{MARKS}_{\text{inv}}[B]$:

```
remove X from BOUNDARY
```

The first step in refining this removal operation is similar to that of the "any element" operation:

> ***If a collection is represented as a sequential collection, an element may be removed by removing the item at the location of the element in the collection.***

When this rule is applied, the following description results:

```
remove the item at location L of BOUNDARY
        L is the location of X
```

Normally, determining the location of an element in a sequential collection involves some kind of search for that location. In this case, however, the location is already known, since X was determined by taking the element at the front of BOUNDARY. The following rule enables us to take advantage of this predetermined knowledge:

> ***If an element X was determined by retrieving the element at location L of a sequential collection C, then L is the location of X in C.***

Testing the condition of this rule involves tracing back over the steps that produce the particular element X and determining that, indeed, the location of X in BOUNDARY is the front. When this is done, the rule can be applied, and we have the following description:

```
remove the item at the front of BOUNDARY
```

From this point on, the program construction process is relatively straightforward, and similar to the "any element" derivation. The end result is the following LISP code:

```
(RPLACD   (CAR (CDR MARKSlisp))
          (CDR (CDR  (CAR (CDR MARKSlisp))))))
```

2.5. UNEXPLORED

The UNEXPLORED collection contains all of those vertices to which no path has yet been found. The only operations applied to this collection are membership testing and addition and deletion of elements. Note that each of these operations is applied to some particular element in the collection. (By contrast, the selection of "any" element of a collection does not have this property.) For such operations, it is often convenient to use a different representation than simply storing the elements in a common structure (as was done with the BOUNDARY collection). In particular, UNEXPLORED will be represented as an array of Boolean values, where the entry for index *k* is TRUE if and only if vertex *k* is in the UNEXPLORED collection.

2.5.1. Representation of UNEXPLORED

The initial description of UNEXPLORED is the same as that of BOUNDARY:

```
UNEXPLORED:
    COLLECTION (integer)
```

One view of collections is simply as a mapping of items to Boolean values. An item maps to TRUE if and only if it is in the collection. This possibility is available through the use of the following rule:

A collection may be represented as a Boolean mapping.

The following description results from applying the rule:

```
UNEXPLOREDmap:
    MAPPING (integer → {TRUE,FALSE})
```

Having decided to use a Boolean mapping, all of the rules available for use with general mappings are applicable here. In particular, the same sequence of rules that was applied to derive the representation of SUCCESSORS can be applied here:

A mapping may be represented explicitly.

UNEXPLORED$_{map}$:
EXPLICIT MAPPING (integers → {TRUE,FALSE})

An explicit mapping may be stored in a single structure.

UNEXPLORED$_{map}$:
STORED MAPPING (integers → {TRUE,FALSE})

A stored mapping with typical domain element X and typical range element Y may be represented with an association table whose typical key is X and whose typical value is Y.

UNEXPLORED$_{table}$:
ASSOCIATION TABLE (integers → {TRUE,FALSE})

An association table whose typical key is an integer from a fixed range and whose typical value is Y may be represented with an array with typical entry Y.

UNEXPLORED$_{array}$:
ARRAY ({TRUE,FALSE})

An array may be represented directly as a LISP array.

UNEXPLORED$_{lisp}$:
LISP ARRAY ({TRUE,FALSE})

Thus, the final LISP representation of the UNEXPLORED collection is an array of Boolean values, with the value being TRUE if the index is in the UNEXPLORED collection and FALSE otherwise.

2.5.2. Remove Y from $\text{MARKS}_{\text{inv}}$["UNEXPLORED"]

The implementation of the "change MARKS[Y] from *U* to *B*" operation involves removing Y from the UNEXPLORED collection. Recall that removing an element from the BOUNDARY set involved modifying one of the links in the list structure representation. As UNEXPLORED is represented differently, removing an element will require a different implementation. The abstract description is the same as in the case of removing X from BOUNDARY.

```
remove Y from UNEXPLORED
```

The first refinement rule for this removal operation is based on the representation of UNEXPLORED as a Boolean mapping.

> ***If a collection is represented as a Boolean mapping, the operation of removing an element X from the collection may be implemented as the operation of changing the image of X from TRUE to FALSE under the mapping.***

When this rule is applied, the following description results:

$$\boxed{\text{change UNEXPLORED}_{\text{map}}[\text{Y}] \text{ from TRUE to FALSE}}$$

The next refinement rule is dependent on the tabular representation of UNEXPLORED:

> ***If a mapping is stored as an association table, the image of a domain element X may be changed from Y to Z by storing Z as the table entry for X.***

(Note that this rule does not use the fact that the old image of X is Y; in tabular representations, storing a value simply overwrites the old value.) When this rule is applied, we have the following description:

$$\boxed{\text{store FALSE in UNEXPLORED}_{\text{table}} \text{ as the entry for the key Y}}$$

The next rule depends of the representation of UNEXPLORED as an array:

> ***If an association table is represented as an array, a value may be stored as the entry for a key K by storing it in the array under the index X.***

Applying this rule, we have:

```
store FALSE in UNEXPLORED_table as the entry for the index Y
```

The last rule to apply here is a LISP-specific rule:

> ***If an array is represented as a LISP array, a value may be stored under an index X by applying the function SETA.***

When this rule is applied, and the LISP representation of FALSE as NIL is used, we have the final LISP code for removing Y from UNEXPLORED:

```
(SETA  UNEXPLORED_lisp  X  NIL)
```

2.6. Final program

The other aspects of the implementation of the Reachability Algorithm are similar to those we have seen. The following is a paraphrase of the final program. (Here, X[Y] denotes the Y^th^ entry in the array X and X:Y denotes the Y field of the plex X.)

Reachability Program

```
VERTICES ← input a list of integers;
SUCCESSORS$_{input}$ ← input an association list of <integer, list of integers> pairs;
SUCCESSORS ← create an array of size 100;
for all X in the list SUCCESSORS$_{input}$:
        SUCCESSORS[X:DOMAIN] ← X:RANGE;
START ← input an integer;
MARKS:EXPLORED ← create an empty list with header cell;
MARKS:BOUNDARY ← create an empty list with header cell;
MARKS:UNEXPLORED ← create an array of size 100;
for all X in the list VERTICES:
        MARKS:UNEXPLORED[X] ← TRUE;
UNEXPLORED[START] ← FALSE;
insert START at front of MARKS:BOUNDARY;
loop:
        if MARKS:BOUNDARY is the empty list then exit;
        X ← front element of MARKS:BOUNDARY;
        remove front element of MARKS:BOUNDARY;
        insert X at front of MARKS:EXPLORED;
        for all Y in the list SUCCESSORS[X]:
                if MARKS:UNEXPLORED[Y] then
                        MARKS:UNEXPLORED[Y] ← FALSE;
                        insert Y at front of MARKS:BOUNDARY;
        repeat;
output MARKS:EXPLORED.
```

Below is given the final LISP code for the program, exactly as it was produced by PECOS:

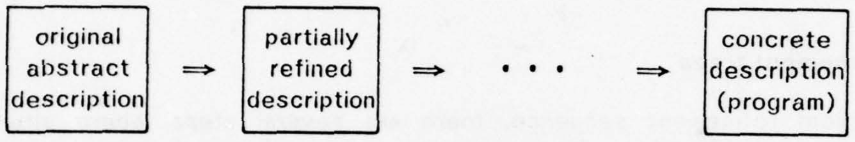
```
(REACH
 [LAMBDA NIL
  (PROG (V0030 V0031 V0032 V0033)
      (PROGN (SETQ V0030 (PROGN (PRIN1 "Points:")
                     (READ)))
          (SETQ V0031 (PROG (V0077 V0075 V0074 V0071 V0070)
                     (PROGN (PROGN (SETQ V0074 (PROGN (PRIN1 "Links:")
                                        (READ)))
                             (SETQ V0070 (ARRAY 100)))
                         (SETQ V0077 V0074))
                  G0079
                     [PROGN (SETQ V0075 V0077)
                         (COND
                          ((NULL V0077)
                           (GO L0078)))
                         (PROGN (PROGN (SETQ V0071 (CAR V0075))
                                 (SETA V0070 (CAR V0071)
                                    (CDR V0071)))
                             (SETQ V0077 (CDR V0077]
                     (GO G0079)
                  L0078
                     (RETURN V0070)))
          (SETQ V0032 (PROGN (PRIN1 "Starting point:")
                     (READ)))
          [SETQ V0033 (CONS (ARRAY 100)
                     (CONS (CONS (QUOTE "HEAD")
                            (QUOTE NIL))
                        (CONS (QUOTE "HEAD")
                            (QUOTE NIL]
          (PROG (V0040 V0038 V0037 V0034)
             (PROGN (SETQ V0037 V0030)
                 (SETQ V0040 V0037))
           G0042
             [PROGN (SETQ V0038 V0040)
                 (COND
                  ((NULL V0040)
                   (GO L0041)))
                 (PROGN (PROGN (SETQ V0034 (CAR V0038))
                         (SETA (CAR V0033)
                             V0034 T))
                     (SETQ V0040 (CDR V0040]
             (GO G0042)
           L0041
            (RETURN))
          (PROGN [PROG (V0044)
                 (SETQ V0044 (CAR (CDR V0033)))
                 (RPLACD V0044 (CONS V0032 (CDR V0044]
             (SETA (CAR V0033)
                 V0032 NIL)))
```

```
G0066
  [PROG (V0054 V0046)
      (SETQ V0054 (CAR (CDR V0033)))
      (COND
       ((NULL (CDR V0054))
        (GO L0045))
       (T (PROGN (SETQ V0046 (CAR (CDR V0054)))
           (PROGN [PROGN [PROG (V0048)
                     (SETQ V0048 (CDR (CDR V0033)))
                     (RPLACD V0048 (CONS V0046 (CDR V0048]
                  (PROG (V0051 V0052)
                     (SETQ V0051 (CAR (CDR V0033)))
                     (SETQ V0052 V0046)
                     (RPLACD V0051 (CDR (CDR V0051]
               (PROG (V0063 V0061 V0060 V0055)
                  (PROGN (SETQ V0060 (ELT V0031 V0046))
                     (SETQ V0063 V0060))
                G0065
                 [PROGN (SETQ V0061 V0063)
                     (COND
                      ((NULL V0063)
                       (GO L0064)))
                     (PROGN [PROGN (SETQ V0055 (CAR V0061))
                           (COND
                            ((ELT (CAR V0033)
                               V0055)
                             (PROGN [PROG (V0057)
                                   (SETQ V0057 (CAR (CDR V0033)))
                                   (RPLACD V0057 (CONS V0055 (CDR V0057]
                                (SETA (CAR V0033)
                                   V0055 NIL]
                        (SETQ V0063 (CDR V0063]
                 (GO G0065)
               L0064
                 (RETURN]
  (GO G0066)
L0045
  [PRINT (CDR (CDR (CDR V0033]
  (RETURN])
```

3. A REFINEMENT MODEL OF PROGRAM SYNTHESIS

3.1. Refinement sequences

PECOS's organization and rule representation are based on a model of program synthesis as gradual refinement. The process may be simply illustrated as a sequence of program descriptions:


original abstract description
⟹
partially refined description
⟹
• • •
⟹
concrete description (program)

Each description in the sequence is slightly more refined (concrete) than the previous description. The first is the program description in the specification language and the last is the fully implemented program in the target language. Such sequences will be referred to as *refinement sequences* and the individual descriptions will be referred to as *program descriptions*. The modification involved in deriving one description from the previous one will be referred to as a *refinement step*[10].

Note that the refinements at each step involve only very small changes to a particular part of the program description. Although such small steps are not required by the formalism, this is one ramification of the attempt to identify and isolate individual programming decisions. To take a simple example, when PECOS implements a membership test using the LISP function MEMBER, there are about a dozen refinement steps, divided about equally between data structure refinements and operation refinements. One could imagine producing such a simple function call in a single step. Indeed, this may be appropriate for some uses. However, in so doing numerous decisions are collapsed into a single step, thereby leaving many of them only implicit in the fact that a particular implementation has been chosen. When one decides to represent a collection using "the standard LISP list representation," one is, in fact, implicitly making several interrelated decisions at once: that the collection will be represented explicitly, that it will be stored in a single structure, that a sequential arrangement of positions will be used to hold the elements, that links will be used to connect positions, that cells from free storage will be used for the positions, and finally that CONS cells will be used.

Another characteristic of PECOS's refinement sequences is that, in many cases, the order in which particular refinement steps occur is unimportant. All that is necessary is that each part of the original description eventually be refined to the most

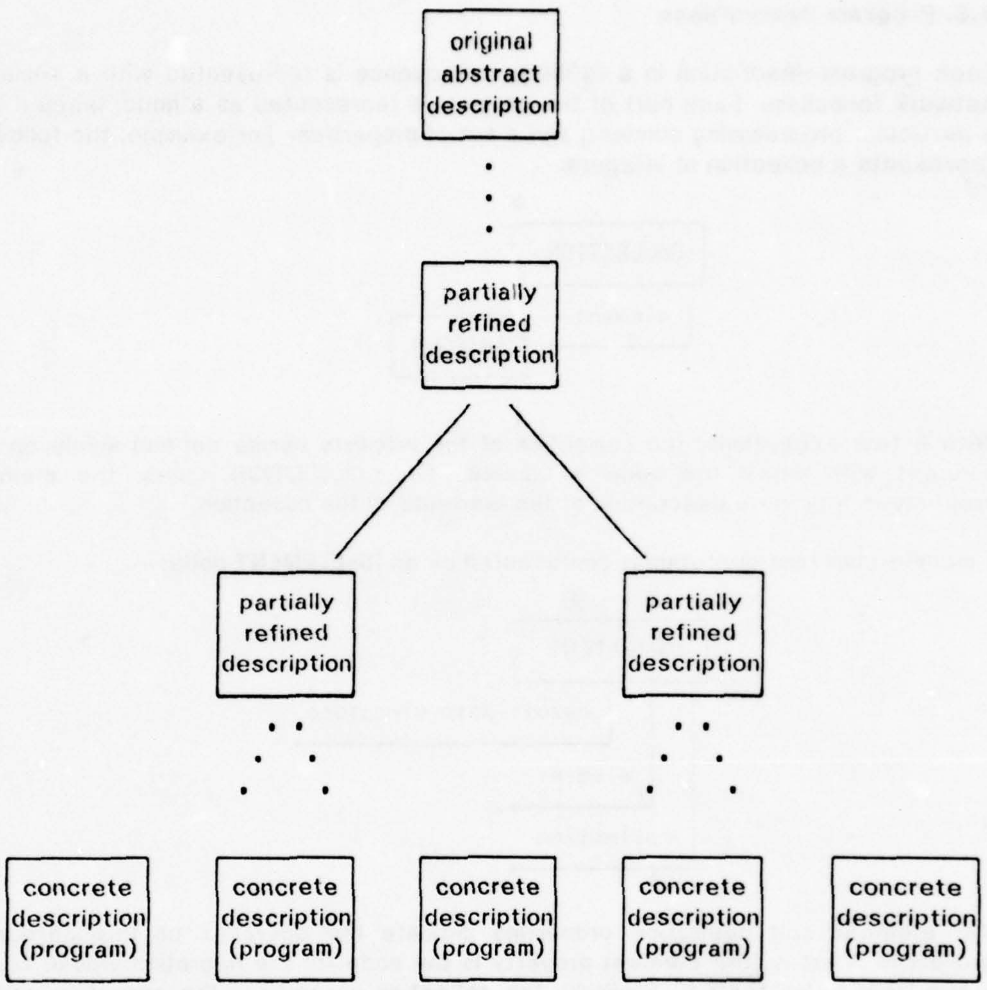
----

10 Section 2 presented several steps in the refinement sequence for the Reachability Program. The entire sequence took almost a thousand steps.

concrete level, the level of the target language. Frequently, separate parts are independent enough that either may be refined before the other. However, there may be a partial ordering on the steps. Typically, data structure refinements precede the refinement of operations on them. Despite such partial orderings, the refinements of particular data structures and operations on them occur as separate steps. This is one of the features that distinguishes PECOS and its knowledge base from most other refinement formalisms. For example, each ALPHARD form consists of a data structure refinement and refinements for each operation on that data structure [Wulf, London and Shaw 1976]. By de-coupling the refinements of data structures and algorithms, greater variability in target programs can be achieved.

3.2. Refinement trees

In a typical refinement sequence, there are several steps where alternative refinements can be made. Thus, the notion of a refinement sequence may be generalized into that of a *refinement tree*, as illustrated below:


original abstract description
partially refined description
partially refined description
partially refined description
concrete description (program)
concrete description (program)
concrete description (program)
concrete description (program)
concrete description (program)

The root of such a tree is the original specification, the leaves are alternative implementations, and each path is a refinement sequence.

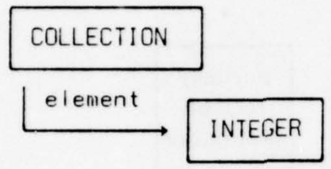
An important feature of such trees is that all the nodes (program descriptions) all represent "correct" programs[11]. Each node represents a step in a path from the abstract specification to some concrete implementation of it. When paths cannot be completed (as happens occasionally), the cause is generally the absence of rules for dealing with a particular program description, rather than any inherent problem with the description itself. The fact that each path is correct greatly facilitates the use of refinement trees as a space to be explored in search of the "best" implementation. This topic is pursued further in section 9.

----

[11] Assuming correctness of the rules, of course.

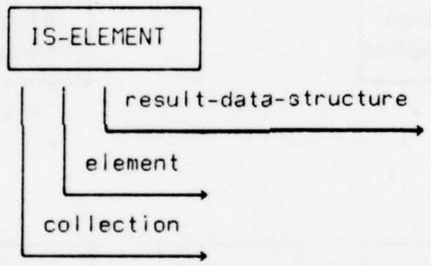
3.3. Program descriptions

Each program description in a refinement sequence is represented with a semantic network formalism. Each part of the program is represented as a node, labeled with a particular programming concept, and a set of properties. For example, the following represents a collection of integers:



With a few exceptions, the semantics of the property names depend solely on the concept with which the node is labeled. For **COLLECTION** nodes, the **element** property is a generic description of the elements of the collection.

A membership test operation is represented by an **IS-ELEMENT** node:



The **element** and **collection** properties indicate the operands of this particular operation. That is, the **element** property is the node for the operation whose result is the item to be tested. Similarly, the **collection** property is the operation whose result is the collection to be tested. The **result-data-structure** property indicates the data structure which is the result of the **IS-ELEMENT** operation. In this case, it would be a **BOOLEAN** node.

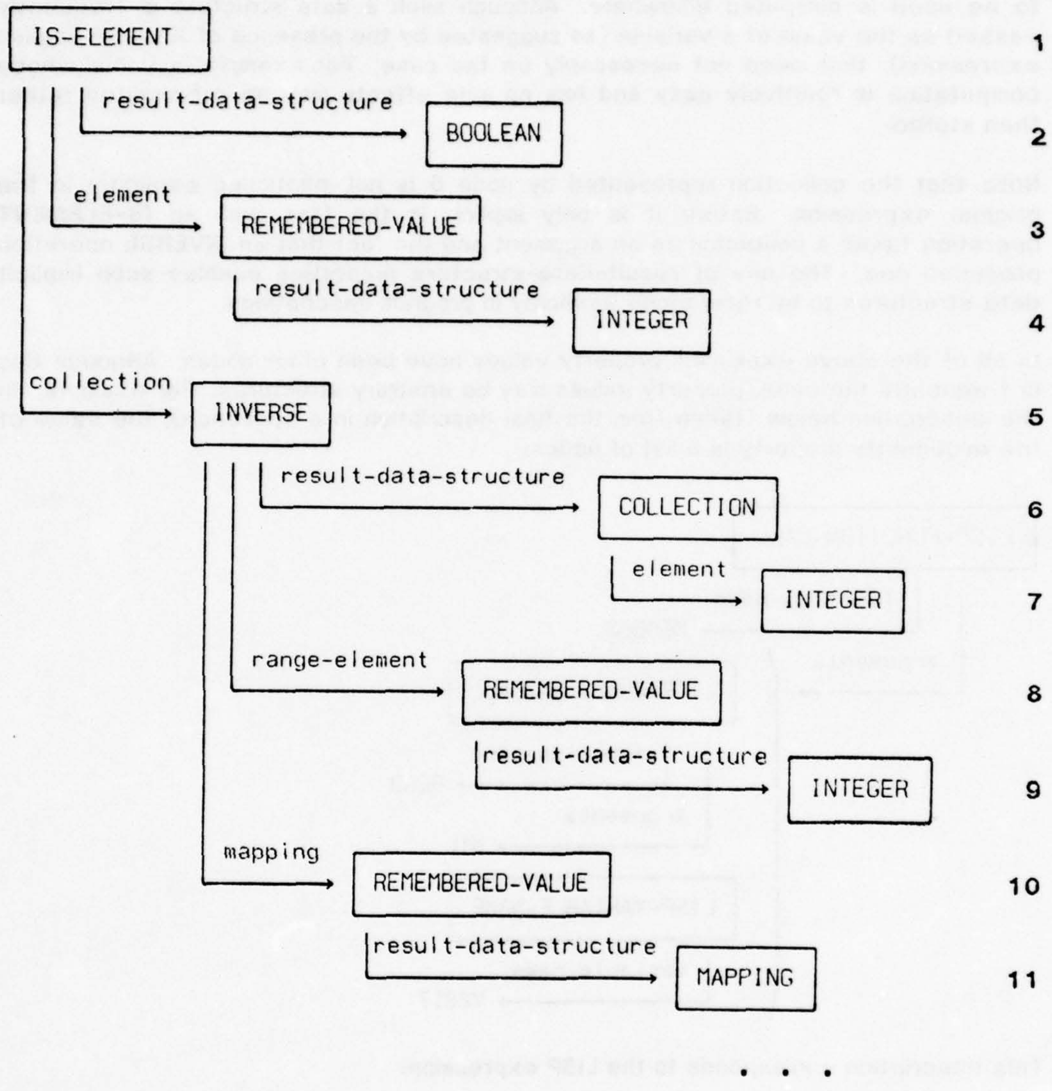
Result-data-structure properties play an important role in PECOS's program descriptions: the node for any operation that produces some result must have such a **result-data-structure** property. The **result-data-structure** property of an operand specifies the data structure that is passed from that operand to the operation. For example, the **result-data-structure** of the **collection** property of an **IS-ELEMENT** node is the node of the collection data structure (as opposed to the operation that produces the collection).

This may be clarified by considering the expression **IS-ELEMENT(X,INVERSE(Y,Z))**[12].

----

[12] "Is X an element of the inverse image of Y under the mapping Z?"; in this example, all primitives involved will be integers.

Part of the node representation for this expression is given below:

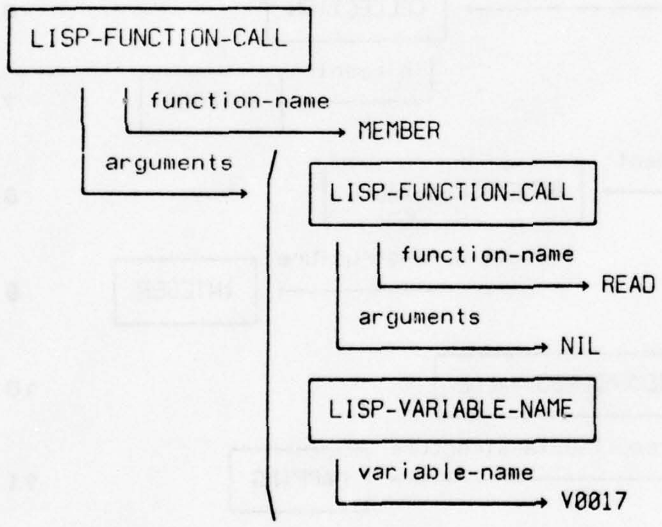


Node 5 is the collection operand of the IS-ELEMENT operation represented by node 1; that is, node 5 represents the expression whose value is the collection to be tested. Node 6, the result-data-structure property of node 5, is the data structure itself, in this case a COLLECTION of INTEGERs. Node 6 thus represents the data structure passed from the INVERSE operation to the IS-ELEMENT operation. As will be seen more clearly later, the fact that a single, particular node represents the data structure passed from one operation to another helps to coordinate the final implementations of the operations so that both are based on the same data structure representation.

Node 3 illustrates another way that data structures may be passed from one operation to another. A **REMEMBERED-VALUE** node signifies that the data structure to be used is computed elsewhere. Although such a data structure is frequently passed as the value of a variable (as suggested by the presence of X in the English expression), that need not necessarily be the case. For example, a value whose computation is relatively easy and has no side effects may be recomputed rather than stored.

Note that the collection represented by node 6 is not mentioned explicitly in the original expression. Rather it is only implicit in the fact that an **IS-ELEMENT** operation takes a collection as an argument and the fact that an **INVERSE** operation produces one. The use of **result-data-structure** properties enables such implicit data structures to be referenced explicitly in program descriptions.

In all of the above examples, property values have been other nodes. Although this is frequently the case, property values may be arbitrary structures. For example, in the description below (taken from the final description in a sequence), the value of the **arguments** property is a list of nodes:



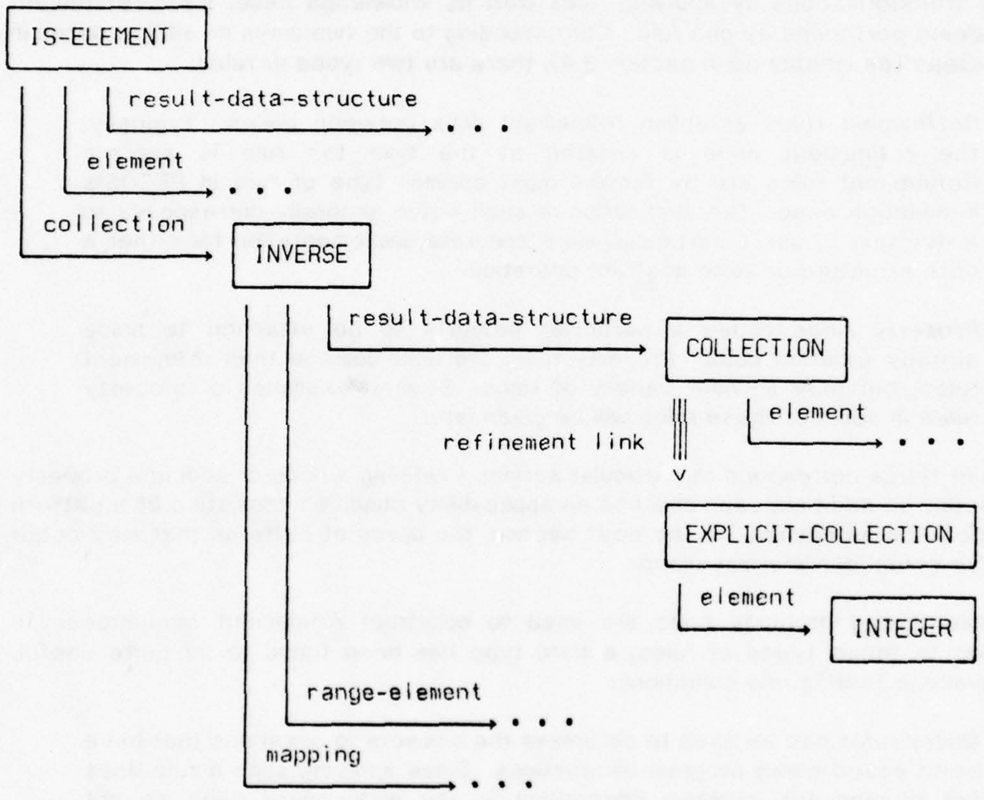
This description corresponds to the LISP expression:

(MEMBER (READ) V0017)

Note that the program descriptions are oriented somewhat toward operations as opposed to data structures. This is no doubt one of many subtle (and often unconscious) effects of using LISP as the target language. One could imagine a data-oriented style in which, for example, data paths could be more easily described than is possible with the program descriptions PECOS uses.

3.4. Refinement steps

In a refinement step, there are two ways that detail can be added to a program description: a property may be added to an existing node, or one node may be identified as a refinement of another. This second type may also involve the introduction of new nodes[13]. Of these two types of changes, the identification of one node as a refinement of another is the most significant. The result of making such a change is illustrated below:



In this refinement step, the **COLLECTION** node has been refined into an **EXPLICIT-COLLECTION** node. In essence, it has been decided to implement the collection explicitly. Note that the element property of the **EXPLICIT-COLLECTION** node is the same node as the element property of the original **COLLECTION** node. Such links between nodes will be referred to as *refinement links*. A sequence of nodes connected by such refinement links will be referred to as a *refinement chain*. Conceptually, such a link may be viewed as a simple replacement of the abstract node by the concrete node.

----

[13] Although one could imagine other types of changes (e.g., modifying a property on a node), no need for them has yet been encountered.

4. RULE REPRESENTATION

4.1. Rule types

As discussed in the previous section, each step in a refinement sequence consists of a small change that transforms one program description into the next. PECOS makes these transformations by applying rules from its knowledge base, each refinement step being performed by one rule. Corresponding to the two ways of adding detail in such steps (as mentioned in section 3.4), there are two types of rules:

> *Refinement rules* establish refinement links between nodes. Typically, the refinement node is created at the time the rule is applied. Refinement rules are by far the most common type of rule in PECOS's knowledge base. The application of such a rule generally corresponds to a decision to use a particular, more concrete, implementation for either a data structure or some abstract operation.
>
> *Property rules* cause a particular property to be attached to some already existing node. Property rules are less common than refinement rules, but play a wider variety of roles. Several examples of property rules in some of these roles will be given later.

The rule types correspond to particular actions - refining a node or adding a property to a node. In addition, each rule has an applicability condition consisting of a pattern of nodes and properties. In the next section, the types of patterns that may occur in these conditions are discussed.

Refinement and property rules are used to construct refinement sequences. In addition to these types of rules, a third type has been found to be quite useful, especially in testing rule conditions:

> *Query rules* can be used to determine the answers to questions that have been posed about program descriptions. Since applying such a rule does not change the program description in any way, query rules do not correspond to refinement steps.

For example, there are about five query rules that determine whether or not two data structures have matching representations. In many situations, it would have been equally possible to embed the query rules directly into the rule conditions that pose the queries. Since there are often several rules for the same query, a facility for "or" conditions would have to be added to the pattern matcher in order to accomplish this. However, query rules do not just simplify the pattern matcher; they also greatly increase the modularity of the individual rules. If a new way of answering a query is discovered, it can simply be added as a new query rule. Otherwise, the conditions of all of the rules that pose the query would have to be modified.

PECOS's representation for these three rule types is shown below:

(REF← <node pattern> <refinement specification>)

(PROP← <property name> <node pattern> <property value>)

(QUERY← <query pattern> <query answer>)

(where **REF←**, **PROP←**, and **QUERY←** are tags indicating the rule type[14]). A **REF←** rule specifies that if a node matches **<node pattern>**, then it may be refined into **<refinement specification>**. A **PROP←** rule specifies that if a node matches **<node pattern>**, then **<property value>** may be attached to that node as the value of the **<property name>** property. A **QUERY←** rule specifies that if a query matches **<query pattern>**, then the answer to the query is **<query answer>**. In the rest of this section, the matching of **<node pattern>**s against nodes in program descriptions will be discussed. The action parts of the rules will be discussed in connection with PECOS's control structure.

4.2. The pattern matcher

In many respects, PECOS's pattern matcher is similar to most other pattern matchers (e.g., QLISP [Wilber 1976], PLANNER [Hewitt 1972]). There are facilities for following various links through the substructures of the pattern and object being matched, for testing conditions on the structures, and for binding variables. In this case, the substructure links are primarily the node properties and refinement links. But since the pattern matcher has been designed with a particular purpose in mind - the codification of programming rules based on the refinement model of program synthesis - there are many idiosyncratic constructs. One of these constructs represents a major departure from most other pattern matchers: in addition to returning with success or failure, a match attempt may also result in an incomplete match. The determination of incomplete matches will be discussed further in section 4.2.2.

4.2.1. Pattern types in rule conditions

A **<node pattern>** consists of a concept name and a list of subpatterns. A **<node pattern>** matches a node in a program description if (1) the concept of the **<node pattern>** is the same as the concept of the node and (2) the subpatterns all match. There is a variety of different types of subpatterns. For ease in implementation, these are generally indicated by the first element of the pattern. Appendix 1 gives a complete list of these types. Some of the more common types will be illustrated by considering several rules in full detail.

----

[14] The ←'s are historical artifacts with no relationship to assignment.

Rule:

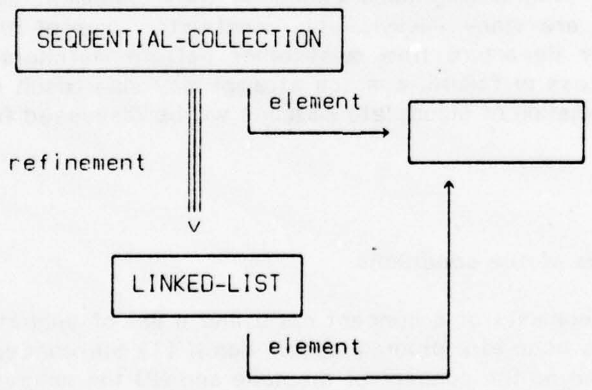
A sequential collection with typical element X may be refined into a linked list with typical element X.

Representation:

```
(REF←    (SEQUENTIAL-COLLECTION
             (#P ELEMENT (←← X)))
         (#NEW LINKED-LIST
             (←#P ELEMENT X)))
```

This rule can be used to refine an abstract data structure (a sequential collection) into a more concrete data structure (a linked list); that is, a refinement link is set up between the abstract node and the concrete node. **REF←** denotes a refinement rule. The first part of the **<node pattern>**, **SEQUENTIAL-COLLECTION** specifies the concept of the node to be refined. **#P** indicates that a particular property should be matched against a subpattern. In this case, the property is the **ELEMENT** property and the subpattern is **(←← X)**, which specifies that the property's value is to be bound to the variable X for use in executing the rule's action. **#NEW** in the **<refinement specification>** specifies that a new node is to be created. **LINKED-LIST** specifies the concept of the new node. **←#P** specifies that a property is to be attached to this node. In this case, the property is the **ELEMENT** property and the value is the value of the variable X (which had been bound to the **ELEMENT** property of the **SEQUENTIAL-COLLECTION** node while evaluating the condition of the rule).

Thus, applying this rule would produce the refinement link indicated in the diagram below:



Note that there are no conditions on the node that represents the elements of the sequential collection, and that the same node is used to represent the elements of the linked list (accomplished using the **←←** mechanism for binding the variable X).

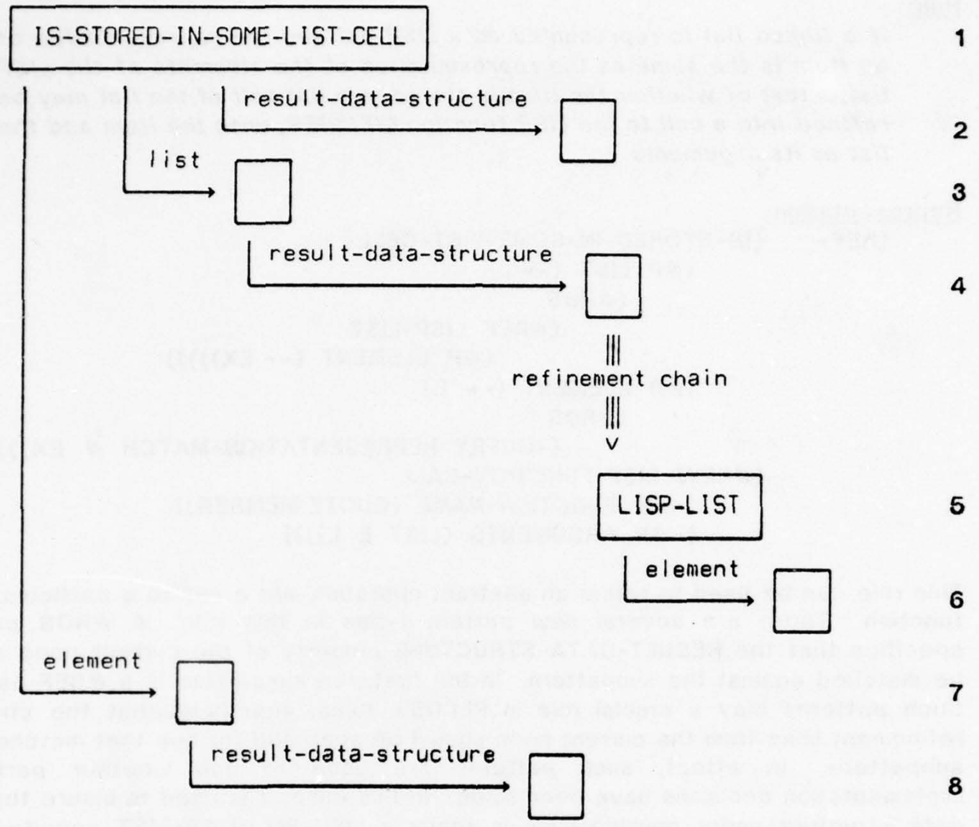
Rule:

If a linked list is represented as a LISP list and the representation of an item is the same as the representation of the elements of the LISP list, a test of whether the item is stored in a list cell of the list may be refined into a call to the LISP function MEMBER, with the item and the list as its arguments.

Representation:

```
(REF←    (IS-STORED-IN-SOME-LIST-CELL
              (#P LIST (←← L)
                   (#RDS
                        (#REF LISP-LIST
                             (#P ELEMENT (←← EX)))))
              (#P ELEMENT (←← E)
                   (#RDS
                        (?QUERY REPRESENTATION-MATCH # EX))))
          (#NEW LISP-FUNCTION-CALL
              (←#P FUNCTION-NAME (QUOTE MEMBER))
              (←#P ARGUMENTS (LIST E L))))
```

This rule can be used to refine an abstract operation into a call to a particular LISP function. There are several new pattern types in this rule. A **#RDS** pattern specifies that the **RESULT-DATA-STRUCTURE** property of the current node should be matched against the subpattern. In the first, the subpattern is a **#REF** pattern. Such patterns play a crucial role in PECOS's rules, specifying that the chain of refinement links from the current node should be searched for one that matches the subpattern. In effect, such patterns are used to ask whether particular implementation decisions have been made. In this case, it is used to insure that the data structure under consideration is really a LISP list (**LISP-LIST** specifies the concept of the node pattern); if it were not, then the rule would clearly be inapplicable. In the second **#RDS** case, the subpattern is a query, (**?QUERY REPRESENTATION-MATCH # EX**). The **#** in the query refers to the current node, in this case the node representing the element being tested. Thus, the query checks whether the element being tested is represented in the same way as the elements of the list. This rule would match the node pattern illustrated below (if nodes 6 and 8 have matching representations):



Note that there are no conditions on nodes **3** and **7**, the operations that produce the argument data structures. The way in which the LISP list is produced has no relevance for this rule. All that matters is that the data structure (node 4) be a LISP list and that the two element nodes (6 and 8) have matching representations. It should also be pointed out that refinement rules automatically cause **RESULT-DATA-STRUCTURE** properties to be inherited. Applying this rule would create a **LISP-FUNCTION-CALL** node with arguments as specified above. In addition, the **RESULT-DATA-STRUCTURE** property of this node would be node **2** above.

Rule:

If the memory scheme of a local memory unit is to bind a variable to the value, a retrieval of the value may be refined into a retrieval of the value of the variable.

Representation:

```
(REF←   (REMEMBERED-VALUE
            (#P LABEL
                (#GLOBAL
                    (#P SCHEME (?#= VALUE-OF-VARIABLE))
                    (#P VARIABLE (←← X)))))
        (#NEW GET-ASSIGNED-VALUE-OF-VARIABLE
            (←#P VARIABLE X)))
```

This rule can be used to refine the abstract operation of retrieving some previously computed value into the operation of retrieving the value of a variable. A global association list is used to relate labels to their **LOCAL-MEMORY-UNIT** nodes. The **#GLOBAL** pattern finds the node associated with this label. The **?#=** pattern is used to test whether the node's **SCHEME** property is exactly equal to **VALUE-OF-VARIABLE**.

Rule:

A memory scheme for a local memory unit is to bind a variable to the computed value.

Representation:

```
(PROP←  SCHEME
        (LOCAL-MEMORY-UNIT)
        (QUOTE VALUE-OF-VARIABLE))
```

This rule, which can be used to attach the **SCHEME** property to a **LOCAL-MEMORY-UNIT** node, is an example of a property rule used to focus a particular decision. The **SCHEME** property specifies the technique that will be used to store and retrieve the particular value. Each **REMEMBER** or **REMEMBERED-VALUE** operation is dependent on this scheme (as illustrated by the previous rule). Thus, the scheme guarantees that all of these operations will be refined in coordination. An alternative way for the rules to deal with such situations could have been to allow any one of the operations to be refined first, and then to force each of the others to be coordinated with the first. There are two primary motivations for the scheme used in PECOS. The first is practical: there is no need to keep extra property links for each operation to refer to the other operations. The second is philosophical: by forcing the operations all to be coordinated through a single property on a single node, a more global view can be taken when selecting one such property rule over another. The use of the scheme property focuses the decision into a single, easily identifiable place. The import of such rules is that a particular programming decision has been identified and isolated.

Rule:

One way to get a variable name is to invent a new one.

Representation:
```
(PROP←  VARIABLE
        (LOCAL-MEMORY-UNIT
             (#P SCHEME (?#= VALUE-OF-VARIABLE)))
        (GENSYM (QUOTE V)))
```

This rule is used to attach the VARIABLE property to a LOCAL-MEMORY-UNIT node. It illustrates the use of property rules to gather further information: a variable name is required, so one is selected. This is, in fact, the only rule that PECOS has for selecting variable names[15]. One could argue that the selection of a particular variable name is also an identifiable programming decision, rather than simply an information gathering task. In fact, when one is concerned about the use of mnemonic variable names, the choice can be important. This has not been a concern with the PECOS implementation.

Rule:

If there is an ordering relation for the elements of a collection, the elements may be stored in the collection according to that relation.

Representation:
```
(PROP←  ORDERING
        (SEQUENTIAL-COLLECTION
             (#P ELEMENT
                  (←← REL #ANSWER
                       (?QUERY ORDERING-RELATION #))))
        (LIST (QUOTE ORDERED) REL))
```

This rule may be used to determine the order in which elements in a collection are to be stored. (The #ANSWER in the ←← pattern denotes the answer of the query about whether there is an ordering relation on the elements of the sequential collection.) This rule illustrates the use of a property rule to further specify a particular data structure without making a refinement. Thus, all of the rules for refining sequential collections (into linked lists and arrays) are applicable whether or not this rule has been applied. It would have been quite possible to have this be a refinement rule, but then duplication of some of the knowledge in the refinement rules for non-ordered sequential collections would have been required. There are several other places where there seemed to be the possibility of using either a refinement rule or a property rule, and the decision has generally been made on the basis of "what seems right at the time." No better justification has been found.

----

[15] This is one of the reasons that PECOS's programs often seem unreadable!

Rule:

If the elements of a collection are to be enumerated in the same order as that in which they are stored, then the enumeration order is the same as the stored order.

Representation:

```
(QUERY← (STORED-ENUMERATION-ORDER
          (#← #
              (#P ENUMERATION-ORDER (←← EO))
              (#P COLLECTION
                  (#RDS
                       (#REF SEQUENTIAL-COLLECTION
                             (#P ORDERING (?#=* EO)))))))
        T)
```

This rather trivial rule can be used as one way of answering a query of the form (?QUERY STORED-ENUMERATION-ORDER n) when the enumeration order is specified to be based on some ordering relation. (Another rule that answers the same query is applicable when the enumeration order is specified to be the stored order, regardless of any ordering relation on the elements.) The #← # in the pattern simply allows multiple patterns to be specified, in this case two #P patterns. The ?#=* pattern tests whether the property's value is the same as the result of evaluating the expression; in this case, the variable EO has been bound previously to the enumeration order of the enumeration node.

Rule:

If a node has been refined into a LISP syntactic entry, the code for the node may be determined by examining the LISP-CODE property of the node, unless the attempt to determine the LISP-CODE property failed.

Representation:

```
(QUERY← (CODE
          (#REF (?CONCEPT-CLASS LISP-SYNTACTIC-ENTRY)
                (#P LISP-CODE (←← C))))
        C)
```

This is the rule that guides the entire process of determining the actual code of the program after all of the parts have been refined into specific LISP objects (see section 5.1). Many concepts have a "class" associated with them. (One of these is known as the LISP-SYNTACTIC-ENTRY class.) A ?CONCEPT-CLASS pattern simply tests the class of the node's concept. Currently, few other uses are made of the concept classes.

4.2.2. The matching process

A relatively standard type of pattern matcher is used to match the **<node patterns>** in the rules against nodes in program descriptions. The most interesting aspect of the matcher is that in addition to identifying success and failure, it may also identify an incomplete match[16]. This occurs in situations where parts of the pattern succeed, but for other parts too little information is available to make a definite answer. Such situations can occur in three ways.

#REF patterns

If a particular node has not been refined far enough to give a definitive answer to a #REF pattern, this can result in an incomplete match. The matcher considers all nodes in the refinement chain leading from the node being matched[17]. If any of these match the **<node pattern>**, the matcher succeeds. Otherwise, the matcher performs a quick check to try to eliminate the possibility of extending the refinement chain to a node that matches the **<node pattern>**. If it can eliminate the possibility, the match fails. If the possibility cannot be eliminated, the matcher signifies an incomplete match and indicates that the refinement chain must be refined further. The quick check is performed by considering only the concept of the most refined node in the chain (call it *concept*$_1$) and the concept of the **<node pattern>** (call it *concept*$_2$). The matcher considers all refinement rules for *concept*$_1$ and the concepts of the nodes they would produce if applied. Refinement rules for each of these concepts are then considered, and so on. If there is any chain of refinement rules that leads from *concept*$_1$ to *concept*$_2$, then the matcher signifies an incomplete match and indicates that the refinement chain must be extended.

For example, the rule that refines **GET-ASSOCIATED-VALUE** into **GET-VALUE-AT-ARRAY-INDEX** is dependent on the fact that the **TABLE** argument has been refined into an **ARRAY**. If this data structure has only been refined as far as an **ASSOCIATION-TABLE**, attempting to match this rule results in the examination of a concept chain as described above. In this case, *concept*$_1$ is **ASSOCIATION-TABLE** and *concept*$_2$ is **ARRAY** (the **<node pattern>** is (#REF ARRAY ...)). There are two rules for refining an **ASSOCIATION-TABLE**, one producing an **ARRAY** (only applicable if the keys are integers in a fixed range) and one producing a **HASH-TABLE**. Since *concept*$_2$ can be reached from *concept*$_1$, the original match attempt yields an incomplete match.

Note that the matcher is relatively conservative in how it determines incomplete matches in this situation. By considering the conditions on the rules, in addition to the concepts involved, it might be possible to determine that a particular rule chain is not possible. Or it might be possible to determine that the node produced by such a

----

[16] The use of incomplete matches in PECOS's control structure (discussed in section 5) is similar to the use of state differences in GPS **[Ernst and Newell 1969]**.

[17] This is the main reason that the refinement links are retained instead of simply replacing the abstract node by the refined node.

rule chain would not match the rest of the #REF pattern. In the above case, for example, if the keys of the association table are not integers, the array rule would be inapplicable. To check for such conditions, however, would require considerably more computation than the simple concept chain involves. (In fact, the rule base is organized around such concepts, so that very little computation is required to determine the existence of concept chains.) Note that this technique allows "misjudgements" of only one kind: it is never the case that a match attempt is considered to have failed when there is still a possibility that the given refinement chain can be extended to match the pattern.

#P patterns

Matching a #P pattern always involves considering a particular property of a particular node. If that property is missing, then the match is considered to be incomplete, and the matcher specifies that the property must be determined. As with #REF matches, the matcher is again conservative in determining when such a match should fail. It makes no checks at all to determine whether there are any rules that could produce a value for the property that would match the pattern. Here, too, the only misjudgements are in being too conservative, i.e., not rejecting matches that could theoretically be rejected immediately.

?QUERY patterns

When a ?QUERY pattern is encountered, the matcher checks to see whether the query has already been answered. If not, an incomplete match is specified, with a request that the particular query be answered. Again, no special checks are made to determine whether there is some way that the query could be answered in such a way that the rest of the pattern would succeed.

4.2.3. Patterns not expressible

There are several types of conditions that are not easily expressed using the pattern types currently available. One example was mentioned earlier: there are no facilities for "or" conditions. There are also no facilities for loops in the actions except for a few ways to map down lists. Since the pattern matcher is implemented using LISP's EVAL, it is not hard to include such things in the rules. But the actual inclusion of arbitrary INTERLISP code is contrary to the intent of the rules, and has been avoided for two reasons. The first is a practical one based on the implementation: the ability to identify every form that occurs in a pattern simplifies the separation of the rules into their various parts and their analysis for free variable usage (see section 5.5). The second is philosophical, going back to the original purpose of this research: the explication of programming knowledge. In discussions of such knowledge, a consistent set of conceptual primitives should be used, and the use of "arbitrary LISP expressions" tends to bypass such primitives. After identifying the conceptual primitive behind the LISP expression, a pattern type for it could be added to the matcher. If the concepts behind the code cannot be identified, then the content of the rule is not sufficiently understood by the rule writer, and the rule should probably be reconsidered informally before trying to express it formally.

Another type of pattern that is not conveniently expressible involves the flow of data between operations. Consider, for example, the refinement of an operation for modifying a linked list by deleting the cell before a cell computed by a complex expression. At the abstract level, the operation looks about like: (DELETE-CELL-BEFORE-CELL <expression>). But a straightforward refinement of such a structure into a call to the LISP function RPLACD yields (RPLACD <expression> (CDR <expression>)), which might involve recomputing the expression. The problem is that the DELETE-CELL-BEFORE-CELL operation implicitly performs a kind of λ-binding, after computing its argument. In order to reflect this, the rule for refining DELETE-CELL-BEFORE-CELL actually produces a COMPOSITE node with a LOCAL-MEMORY-UNIT in which to store the argument. While this rule correctly reflects the semantics of what is involved, it seems overly ponderous to try to express it. A better facility for dealing with patterns of data flow would be a useful extension.

4.3. Idiosyncrasies of the rule formalism

By way of clarification, a few of the rule formalism's idiosyncrasies should be discussed before they appear in the rules given in section 6. One of these involves the refinement of operations which are applied to data structures that have already been refined. As a typical case, consider a membership test applied to a collection, X. The complete refinement of X into a LISP list involves six rule applications, each further specifying the representation of X. While this is straightforward, there is more difficulty with the membership operation: how many refinement steps should be involved? With a single step (say, into the LISP function MEMBER) many other alternatives are missed. For example, if the list is ordered, some efficiency can be gained by abandoning the search for the element when some larger element is found. The knowledge that leads to this more efficient algorithm also applies if the collection is represented as an ordered array. This suggests that there may be some intermediate concept between "membership test" and "MEMBER function", perhaps "membership test for sequential collection". In PECOS's rules, this argument has been carried to an extreme, and there is a separate named concept for "membership test" corresponding to each data structure refinement level. Thus, membership test operations typically go through a sequence of refinements that parallel those of the data structure to which the test is applied. In retrospect, it is not at all clear that such a complete set of intermediate concepts is appropriate: is there really any content to a concept like "membership test in an explicit collection"? In the rule descriptions in section 6, many of these intermediate-level operations will be omitted.

A related issue concerns the way that the data structure and operation refinement rules are related to each other. In most refinement formalisms, operation refinement rules are attached directly to the data structure refinement rules (e.g., *forms* in ALPHARD [Wulf, London and Shaw 1976]). In PECOS, they have been totally separated. One consequence of this is that the rules for refining an operation typically have conditions testing the refinement chains of its operands. These operand checks are performed by #REF patterns. As discussed in section 4.2, a #REF pattern specifies that the refinement chain from the current node be searched for a node that satisfies the rest of the pattern. As certain concepts may be linked through different refinement chains, it is often necessary to complicate #REF patterns to distinguish the refinement chains from each other. For example, COLLECTION may be refined into STORED-COLLECTION in two different ways:

```
COLLECTION     =>  EXPLICIT-COLLECTION      =>  STORED-COLLECTION
element: X         element: X                   element: X

COLLECTION     =>  CORRESPONDENCE           =>  STORED-COLLECTION
element: X         domain: X                    element: <X, value>
                   range: boolean values
```

Note that (#REF STORED-COLLECTION) would match both chains. The primary technique for dealing with such multiple refinement chains is the use of "representation match" queries. Thus, since the **element** properties of the two STORED-COLLECTION nodes would be different, a pattern like

```
(#REF STORED-COLLECTION
      (#P ELEMENT
          (?QUERY REPRESENTATION-MATCH # X)))
```

would only succeed when the **element** property matched X, effectively distinguishing the two cases. Such representation match patterns play a very important role, although their necessity only became apparent after enough rules had been added to make such multiple refinement chains possible[18].

In part, the inclusion of such details in the rule conditions is a consequence of the decision to separate the programming knowledge into "independent" chunks. Such conditions often merely test whether a particular rule has been applied by testing whether the situation matches what would have resulted had the rule been applied. A more efficient codification of the knowledge might be to link the rules together more directly, either by combining several rules into a single unit or by allowing rules to reference other rules by name.

----

18 In fact, the omission of such conditions was a major source of bugs in the earlier rules.

5. PECOS'S CONTROL STRUCTURE

5.1. An agenda of tasks

As noted above, PECOS derives refinement sequences by successively retrieving and applying rules from its knowledge base. This process is guided by an agenda of tasks[19]. Each task specifies some action that must be performed before the refinement sequence can be completed. Achievement of a task yields a new description in a refinement sequence.

A task is achieved by applying a rule (i.e., evaluating the rule's action) that fits the task. ("Fit" will be defined shortly.) At each stage a task is selected and a rule for that task is retrieved and applied. While retrieving a rule for a task, subtasks may be generated. These are added to the agenda and considered before the original task is reconsidered[20].

There are three types of tasks:

(REFINE n) specifies that node n is to be refined.

(PROPERTY p n) specifies that property p of node n is to be determined.

(QUERY rel arg_1 arg_2 ...) specifies that the query (rel arg_1 arg_2 ...) must be answered.

The first two correspond to the ways that detail may be added in a single refinement step, as discussed in section 3.4. The third amounts to a facility for allowing rules to be used to answer questions about program descriptions. For example, (QUERY STORED-ENUMERATION-ORDER n) is used to determine whether the order of enumerating the elements of a collection is the same as the order in which they are stored. QUERY tasks are typically set up while testing the applicability of a rule to some other task.

There is nothing particularly unusual about PECOS's use of an agenda for its control structure; several other current AI systems use similar techniques [Lenat 1976, Bobrow and Winograd 1977]. There are, however, a few interesting aspects which merit further discussion.

Before adding a subtask to the agenda, PECOS checks whether it can be achieved

----

19 Much of the agenda control structure was developed jointly with Elaine Kant in connection with her work on LIBRA, PSI's efficiency expert.

20 As will be clarified in section 5.4, the order in which tasks are considered is approximately depth-first.

easily. Currently, an "easy" subtask is defined to be a **QUERY** or **PROPERTY** task that has a single applicable rule with no subtasks. In such cases the rule is applied immediately without modifying the agenda at all[21].

Another feature is the method of determining when to add a task to the agenda. Two different schemes have been tried. The first involves adding a task only when it is identified as a subtask of another task on the agenda[22]. In essence, tasks are added to the agenda only "on demand". Initially, the agenda consists of a single task: **(QUERY CODE 1)**[23]. Since node **1** is the top node, the task amounts to a request to determine the code for the entire program. As no rule can answer the query immediately, subtasks are set up and added to the agenda. After these subtasks are considered, the original task is reconsidered, and again subtasks are added to the agenda. Most subtasks thus generated are refinement tasks: the parts of the program must be refined to sufficient detail that code for them can be determined with a single rule application. Generally, this means that they must be refined into any of various kinds of LISP constructs, such as function calls or constants.

The second scheme takes advantage of the fact that PECOS's basic purpose is to refine abstract concepts into specific implementations: whenever a node is created, a refinement task for that node is added to the agenda, regardless of whether or not it is known to be a subtask of some other task. Since most (usually all) parts of the program must eventually be refined into LISP constructs, those tasks can be set up as soon as possible. This technique offers the opportunity for greater flexibility in selecting the tasks to consider next, since the agenda has a more "complete" list of things to do. In the second case, the initial agenda also has a single task: **(REFINE 1)**. Just as in the first scheme, working on a task may still require that other tasks be set up (property and query tasks). The second scheme permits a simple two-pass process to be used to determine the LISP code of the final implementation: when no further tasks remain, a **(QUERY CODE 1)** task is set up and worked on. Since each node has been refined into a LISP construct, the code can be determined without generating any further subtasks. The switch from the first scheme to the second was quite simple (less than one hour's work), largely because of the convenience of using an agenda mechanism.

----

[21] Strictly speaking, this is a feature of the pattern matcher, not the control structure.

[22] Subtask identification is considered further in section 5.3.

[23] **(QUERY CODE n)** specifies that the code for the subpart of the program headed by node **n** must be determined.

5.2. A tree of program descriptions

Recall that the only changes involved in the refinement steps are the identification of one node as a refinement of another and the addition of properties to existing nodes. Each rule application for a **REFINE** or **PROPERTY** task corresponds to such a step. Thus, the task agenda guides the construction of such refinement sequences.

When more than one rule is applicable, separate program descriptions can be generated for use with each rule. The result is a refinement tree such as those discussed in section 3.2. PECOS uses a simple context mechanism to deal with these trees, thereby avoiding the necessity of copying the entire descriptions at each split. The process of finding a particular node or property is then a simple search from the current description through its chain of ancestors. Since agendas are also maintained "relative" to a context, several branches in a refinement tree may be explored in parallel without interfering with each other.

This is perhaps best clarified by an example. A **COLLECTION** node has an **element** property to describe the data structure of the elements of the collection. When a **COLLECTION** node has been refined into an **EXPLICIT-COLLECTION** node, the refined node also has an **element** property (and, in this case, the two properties would have the same value). The property must be attached explicitly to each of the nodes. That is, in fact, precisely what the particular refinement rule does. In effect, the rules specify which properties are to be inherited, and the execution of the rule action forces those inherited properties to be attached explicitly to the new node. Later in the synthesis process, after several descendant contexts have been created, an attempt to access the **element** property of the **EXPLICIT-COLLECTION** node would result in a backward scan through the contexts until one was found in which the **EXPLICIT-COLLECTION** node had that property.

For purposes of efficiency, separate program descriptions are actually set up only at choice points (when more than one rule is to be applied to the same task). When only one rule is to be applied, the same context is used. This saves a considerable amount of overhead that would otherwise be used for establishing contexts and searching context trees. Thus, the refinement trees that PECOS constructs are actually collapsed trees in which each description is either a leaf or has more than one successor.

When no rules are applicable to a task, the task is considered to have failed. However, the refinement process may still continue. For example, failure of a **(QUERY ORDERING-RELATION n_1)** task simply indicates that no ordering relation could be found for the data structure represented by node n_1. The effect of this failure would be that certain rules for enumerating collections of such structures would be inapplicable to some other task (usually a **REFINE** task, say for node n_2). Node n_2 might still be refined by some rule that is not dependent on the existence of such an ordering relation. In other situations, however, task failure may imply the failure of the entire sequence that is currently being followed. PECOS is relatively conservative in deciding to abandon a path: a path is considered to have failed only if a refinement task fails. Under some circumstances, this may require more work to recognize failure than is strictly necessary, but this has not yet been a significant problem.

5.3. Task achievement through rule application

Tasks are achieved by applying a rule that fits the task, where "fit" is defined as follows:

> A (REF← <node pattern> <refinement specification>) rule fits a (REFINE n) task if the <node pattern> matches node n.
>
> A (PROP← <property name> <node pattern> <property value>) rule fits a (PROPERTY p n) task if the <property name> is p and the <node pattern> matches node n.
>
> A (QUERY← <relation name> <pattern$_1$> <pattern$_2$> ...) rule fits a (QUERY rel arg$_1$ arg$_2$ ...) task if the <relation name> is rel and each <pattern$_i$> matches the corresponding arg$_i$.

Matching of a <node pattern> to a node was defined in section 4.2.

While it is relatively simple to implement this scheme, searching and testing every rule for every task would be computationally prohibitive. The organization of PECOS's knowledge base is designed to eliminate the need for most of this search.

5.3.1. Rule retrieval

Recall that a necessary condition for a <node pattern> to match a node is that the two concepts be the same. A simple discrimination net is used to filter out those rules that fail this part of the test. Associated with each rule is a set of task patterns. These are of three forms, related to the types of tasks: (REFINE <concept>), (PROPERTY <property name> <concept>), and (QUERY <relation name>). Associated with each concept is a list of all rules specifying that concept in one of the associated task patterns[24]. This list is partitioned into refinement and property rules. The property rules are further partitioned according to the property name. A similar scheme is used for query patterns. Given a task (and the concept of the node or query relation involved), the associated list of "relevant" rules can easily be retrieved. Using this scheme, the number of rules which must be tested for applicability to any given task is generally quite small (sometimes only one, rarely more than three or four).

----

[24] There are also separate lists for use with <concept class> expressions.

5.3.2. Matching rules to tasks

Once this preliminary filtering is done, the subpatterns of the rule's **<node pattern>** are matched against the properties of the node specified in the task. As described in section 4.2, the result may be success, failure, or an incomplete match. With incomplete matches, the match procedure indicates which parts of the program are to be considered further by specifying a set of tasks. These tasks are added to the agenda and noted to be subtasks of the original task. After considering the subtasks, the original match is reconsidered, and subtasks may again be specified. The cycle repeats until the match succeeds or fails. Suppose, for example, that the task is to refine an **IS-ELEMENT** node. If the **collection** data structure has been refined into a Boolean mapping, the rule for refining an **IS-ELEMENT** node into a **GET-IMAGE** node is applicable. If the collection has been refined into something other than a Boolean mapping, the rule is inapplicable. However, if it has not been refined at all, there is only an incomplete match, with a subtask of refining the collection node. Once the collection node has been refined, the attempt to match the rule to the task is repeated.

Note that the achievement of the subtask does not imply that the original rule will match the original task, but only that the matching process can proceed. In effect, a kind of "generalized subtask" is used: it specifies that a certain part of the program must be considered further, but does not include constraints on what the result must be[25]. In fact, it is frequently the case that the same subtask is specified for several relevant rules and the subtask's achievement eliminates most of the rules. In effect, such subtasks often perform a kind of filtering on the list of relevant rules.

5.3.3. Separation of applicability and binding

As in most pattern-directed inference systems, the matching process performs a dual purpose: the determination of success or failure, and the determination of various bindings to be used when the rule's action part is executed. In PECOS, these two aspects have been separated into distinct stages, termed the *applicability stage* and the *binding stage*. In the applicability stage, only those parts of the pattern that may lead to failure of the entire match are attempted. The binding stage is not attempted until the match has completed the applicability stage successfully without generating any further subtasks. In the binding stage, only those parts of the pattern that return bindings for use in the rule action are considered. Subtasks may be generated in each of these stages.

As an illustration, consider the following rule:

> ***If a value is remembered by storing it as the value of a variable whose name is X, a retrieval of the remembered value may be refined into a retrieval of the value of the variable X.***

----

25 MYCIN uses a similar kind of generalized subgoal, but for different purposes[Davis, Buchanan and Shortliffe 1977].

whose internal form is as follows:

```
    (REF←   (REMEMBERED-VALUE
                (#P LABEL
                    (#GLOBAL
                        (#P SCHEME (?#= VALUE-OF-VARIABLE))
                        (#P VARIABLE (←← X)))))
            (#NEW GET-ASSIGNED-VALUE-OF-VARIABLE
                (←#P VARIABLE X)))
```

Since there is a condition on the **SCHEME** property (namely, that the property be **VALUE-OF-VARIABLE**), that subpattern appears in the applicability part:

```
        (#P LABEL
            (#GLOBAL
                (#P SCHEME (?#= VALUE-OF-VARIABLE))))
```

A **(PROPERTY SCHEME n)** subtask would be generated during the applicability stage. By contrast, there are no conditions on the **VARIABLE-NAME** property and it would appear as a subtask during the binding stage. The binding part is:

```
        (#P LABEL
            (#GLOBAL
                (#P VARIABLE (←← X))))
```

In retrospect, the distinction between the "applicability" and "binding" parts of a pattern has some merit in preventing certain kinds of unnecessary work and in delaying commitment to a particular rule for as long as possible. But the overhead incurred by the total separation employed by PECOS is relatively high. A more efficient technique might involve only a single condition-testing phase in which proposed subtasks are tagged by type (i.e., either "applicability" or "binding").

5.4. Task ordering

There is only one absolute constraint on the order in which tasks must be achieved: a rule may not be applied until it has passed its applicability test and its bindings have been gathered. This ordering is determined through the use of subtasks generated during the applicability and binding processes. Note that this ordering is only partial: either of two unordered parts may be refined before the other in a refinement sequence[26]. Finally, note that there is nothing in the formalism that prevents two tasks from being subtasks of one another. In fact, such cases have occurred while the rules were being debugged (PECOS did not recognize the problem and entered a loop without an exit!).

----

[26] From another perspective, two parts of a program may often be refined independently from each other without interfering with the correctness of the final program. Of course, the determination of the *best* way to refine a particular part may involve looking at many other parts.

However, task ordering can have a significant effect on the size of the refinement tree constructed, as well as on the overall efficiency of the program. For this reason, various strategies have been used in an effort to reduce tree sizes and increase efficiency. These strategies are based on the fact that the process of achieving a task goes through several separable stages:

(a) retrieve rules relevant to the task
(b) test each of these for applicability
(c) select one of these for application
  (or establish separate contexts for each rule application)
(d) determine the bindings required by the rule action
(e) execute the rule action

Stage (a) needs to be done only once and can be accomplished relatively quickly using the discrimination net described earlier. Stage (b) is relatively complex and may need to be repeated several times until all matches either fail or succeed (and no subtasks remain)[27]. Stage (c) is relatively simple except that extra overhead is involved when separate contexts must be established. Stage (d) is generally simple, although occasionally subtasks are generated. Stage (e) is executed only once.

In addition, tasks in several of the stages can be further categorized. Tasks in stage (a) can be distinguished on the basis of whether or not they are known to be subtasks of other tasks. (Recall that refinement tasks are set up as soon as new nodes are created.) Tasks in stages (b) and (d) can be distinguished on the basis of whether or not the stage has already been attempted and subtasks identified. Tasks in stage (c) can be distinguished on the basis of whether or not more than one rule is applicable to the task. In addition, a task may be either *active* or *suspended* depending on whether any of its outstanding subtasks have been achieved. Only active tasks need to be worked on, since considering a suspended task would only yield the same set of subtasks.

Based on these stages and categories, a task can be identified as being in one of the following eight states:

state name	description
new task	stage (a), not known to be a subtask of some other task
get rules	stage (a), known to be a subtask of some other task
test rules	stage (b), not yet attempted
more subtasks	stage (b), subtasks identified
pick rule	stage (c), applicable rules identified
choice point	stage (c), several rules applicable, choice not yet made
gather bindings	stage (d)
apply rule	stage (e)

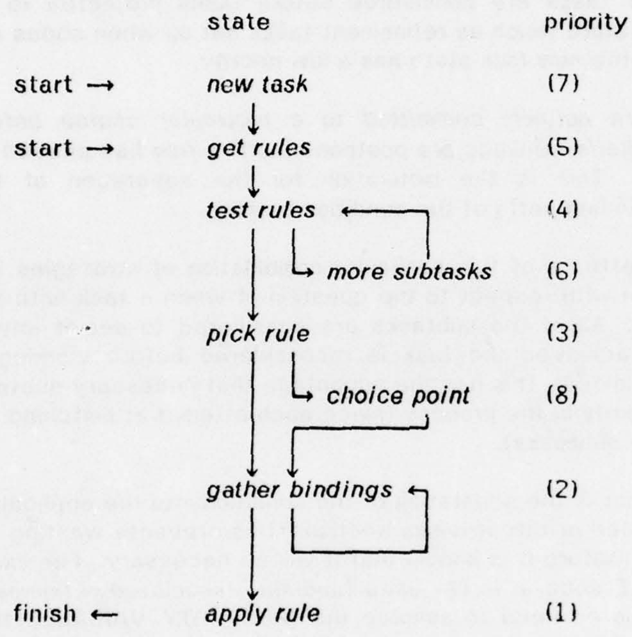
There is no necessity that a single task be carried through all of these states before

----

27 PECOS has a facility for splitting a refinement path as soon as one rule's match has succeeded, but this is currently not being used.

considering any other tasks. The process of working on a task is easily interruptible in each of these states. That is, one task can be carried through some number of these states and then be interrupted for an extended period of time during which many other tasks may be initiated or completed. This observation allows certain gains in efficiency by carefully picking which task to work on next. This selection is based on a priority ranking of these states. During each cycle, PECOS works on one of the tasks in the highest priority state for which there are any active tasks[28].

The progression of a task through its states and the priority rating of each of these states are illustrated below:



The priority ordering shown above incorporates several different strategies, including the following:

> *Put off making decisions for as long as possible, in the hope that they will be easier to make.* Choice points are delayed as long as possible. This allows much of the program to be developed before considering choice points, making more information available when the decision must be made, and saving considerable effort which might otherwise be duplicated in separate refinement sequences if a decision cannot be made and the refinement path must be split. This is the most important strategy, and the reason that the *choice point* state has the lowest priority.

----

[28] In fact, PECOS picks the task that entered that state most recently, but this is an accident of the implementation due to the use of task lists for each state, with tasks being added and removed at the front of the list.

If a task can be completed easily, do it and get it over with. Tasks in advanced stages are considered before tasks in early stages. Hence, the relatively high priorities of the *apply rule* and *gather bindings* states.

If a subtask can be done easily, do it on the assumption that the result may help eliminate some possibilities for the original task. Some consideration is given to all of a task's subtasks before reconsidering the task. So the *get rules* state has a higher priority than *more subtasks*.

First work on things that you know you have to do. Tasks known to be subtasks of other tasks are considered before tasks projected to be necessary in the future (such as refinement tasks set up when nodes are created). Hence, the *new task* state has a low priority.

Wait until you are actually committed to a particular choice before cleaning up the details. Bindings are postponed until a rule has passed its applicability test. This is the motivation for the separation of the applicability and binding parts of the condition testing.

One of the interesting effects of this particular combination of strategies is that a mixed approach is taken with respect to the question of when a task with subtasks should be reconsidered. All of the subtasks are considered to see if any can be done easily; if any is achieved the task is reconsidered before working on any subtasks that involve choices. This has the advantage that necessary subtasks may be identified relatively early in the process (since each attempt at matching a rule to a task may indicate new subtasks).

Another interesting aspect is the separation of the condition into the applicability and binding parts, as described in the previous section. This prevents wasting effort in trying to achieve a task before it is known that it will be necessary. For example, if the **VALUE-OF-VARIABLE** scheme is not used (and the associated refinement rules are not applied), there is no need to achieve the **(PROPERTY VARIABLE-NAME n)** task. This can be especially critical if there are several rules for achieving such a subtask. There is, however, a certain amount of overhead involved in using the separation. It is not clear yet how much efficiency is really gained (or lost) by the technique.

The particular ordering and strategies used in PECOS have been developed empirically, based on observations of PECOS's behavior. The value of many of these strategies is probably derived from characteristics of the types of rules in the knowledge base and on the particular implementation. For example, two of the strategies seem somewhat contradictory at the abstract level. The separation of applicability and binding is based on the assumption that tasks whose necessity is not known should be delayed for as long as possible. The establishment of refinement tasks for every node as it is created is based on the assumption that such tasks should be attempted even before their absolute necessity is known. The utility of these strategies is based on a characteristic of the knowledge base and the tasks involved: refinement tasks are almost always necessary, while other types of tasks frequently are not. Thus, it would be premature to make any generalized claims about the utility of these strategies for other domains.

5.5. Automatic derivation of rule parts

As indicated in the previous section, each rule is used in several different ways in the process of applying it to a particular task. For the sake of efficiency, PECOS maintains separate representations of the rule for each of these uses. These are derived automatically by PECOS and are invisible to the user (and to the rule writer). That is, rules are always dealt with externally in terms of condition-action pairs, as described in section 4. Internally, PECOS maintains a list of relevant task patterns, the applicability subpatterns of the condition, the binding subpatterns of the condition, and a specification of the action to be executed[29]. The derivation of each of these parts is relatively straightforward.

5.5.1. Derivation of relevant task patterns

Relevant task patterns are derived in a simple manner, based on the type of the particular rule. A **REF←** rule has a relevant task pattern of the form **(REFINE <concept>)**. A **PROP←** rule has a relevant task pattern of the form **(PROPERTY <property name> <concept>)**. A **QUERY←** rule has a relevant task pattern of the form **(QUERY <relation name>)**. Currently, each rule has exactly one such pattern.

5.5.2. Derivation of applicability and binding patterns

The most interesting aspect of the derivation of rule parts relates to the separation of a rule's node pattern into applicability and binding parts. The process is driven by a table of pattern types. Each type has tags indicating whether it performs a test (e.g., (#REF MAPPING) tests whether a node has been refined into a MAPPING) and whether it performs any bindings (e.g., (←← X) binds the variable X). A pattern is then included in the applicability part if it or any subpattern performs a test. Similarly, a pattern is included in the binding part if it or a subpattern performs a binding.

The process is complicated somewhat by the fact that some of the patterns use variables freely, so that subpatterns that bind these parts must also be included. In order to deal with this, the pattern tables also indicate which subparts are evaluated, so that these may be checked for their use of free variables. The process that uses these tables examines subpatterns in the opposite order from that in which they will be evaluated at run time in order to maintain an accurate list of all free variables used freely in later subpatterns. Any subpattern that binds a variable on this list must be included. The relevant table entries for each of the pattern types are included in appendix 1.

----

[29] Each of these last three is stored as the compiled definition of a LISP function, so that rules can be matched and executed quickly by evaluating simple LISP expressions.

As an example of this derivation process, consider the following rule:

```
    (REF←   (IS-STORED-IN-SOME-LIST-CELL
                  (#P LIST (←← L)
                      (#RDS
                          (#REF LISP-LIST
                                (#P ELEMENT (←← EX)))))
                  (#P ELEMENT (←← E)
                      (#RDS
                          (?QUERY REPRESENTATION-MATCH # EX))))
            (#NEW LISP-FUNCTION-CALL
                  (←#P FUNCTION-NAME (QUOTE MEMBER))
                  (←#P ARGUMENTS (LIST E L))))
```

Analysis of the action determines that E and L are used freely. Those parts of the condition that bind either of these variables are included in the binding part:

```
                  (#P LIST (←← L))
                  (#P ELEMENT (←← E))
```

Now the applicability part of the condition is constructed. Each of the subpatterns is considered to determine whether it could result in failure or whether it binds any variable used freely in later expressions that are part of the applicability check. The two subpatterns are considered in reverse order:

```
                  (#P ELEMENT (←← E)
                      (#RDS
                          (?QUERY REPRESENTATION-MATCH # EX))))

                  (#P LIST (←← L)
                      (#RDS
                          (#REF LISP-LIST
                                (#P ELEMENT (←← EX)))))
```

While considering the first of these subpatterns, the process is called recursively. This inner call determines that (?QUERY REPRESENTATION-MATCH # EX) could result in failure so it must be included. The fact that EX is used freely is noted. Since a subpattern of the #RDS pattern is included, it must also be included, and so on. The (←← E) need not be included since it cannot result in failure and does not bind a variable used freely in the applicability test. Thus, it is determined that the following must be included in the applicability part:

```
                  (#P ELEMENT
                      (#RDS
                          (?QUERY REPRESENTATION-MATCH # EX)))
```

When the second of the subpatterns is considered, the (←← EX) is included since it binds EX, and the #REF is included since it could result in match failure. Thus, the following must also be included in the applicability check:

```
(#P LIST
    (#RDS
        (#REF LISP-LIST
            (#P ELEMENT (←← EX)))))
```

and the full applicability pattern is then the following:

```
(#P LIST
    (#RDS
        (#REF LISP-LIST
            (#P ELEMENT (←← EX)))))
(#P ELEMENT
    (#RDS
        (?QUERY REPRESENTATION-MATCH # EX)))
```

5.5.3. Multiple-valued matches

Several of the pattern types may succeed in several ways. That is, several different sets of bindings may satisfy a pattern. In such cases, every set of values is considered to be a separate relevant rule. Those variables that can have multiple bindings are considered to be parameters for the rule. The process of retrieving relevant rules is thus somewhat more complicated than indicated in the previous section: after the list of relevant rules has been found, each rule on the list is checked for parameters and all possible values for these parameters are then determined. Very few of the rules in PECOS's current knowledge base are parameterized in this fashion. Those pattern types that yield such parameters are indicated in appendix 1.

6. A KNOWLEDGE BASE OF PROGRAMMING RULES

One of the most important aspects of PECOS's development has been the detailed codification of knowledge about symbolic programming in the form of explicit rules. A formalism for expressing such rules (discussed in section 4) has enabled the statement of the rules in machine-useable form and the development of a system for applying them. However, most of the knowledge embodied in the rules is independent of the particular formalism used to express them. In order to separate the content of the rules from the idiosyncrasies of the formalism the rules discussed in this section will be expressed in English. The translations from internal form are relatively loose. The same internal representation may be stated in several different ways and some details are omitted on the assumption that they are "obvious." Hopefully, this will make them somewhat easier to read and understand, without any significant loss of content. Any implementation of such rules, of course, requires that a great deal of attention be paid to just such details. To indicate the detail required, as well as the way the formalism can be used to express the rules, several rules will also be given in their complete, machine-readable form.

The rules divide naturally into categories based on the particular concepts involved. The rules for three of these categories will be discussed in detail, as summarized below:

Representations of collections
- Boolean mapping
- linked list
- array

Enumeration and sorting of stored collections

Representations of mappings
- collection of pairs
- hash table
- array
- property list entry
- inverted mapping

In addition, aspects of several other rule groups will be discussed.

On the assumption that most readers will not be interested in examining all of the rules in full detail, a brief index of the rule groups is presented at the end of this section.

6.1. Collections

A collection may be thought of as a structure consisting of any number of substructures, each an instance of the same generic description[30]. There are six basic operations for dealing with collections, as given below (with names used in PECOS's specification language):

NEW-COLLECTION
: Creates a new collection and returns it as the operation's value. A list of elements to be contained in the collection initially may also be specified.

ADD-ELEMENT
: Adds a given element to a given collection.

REMOVE-ELEMENT
: Removes a given element from a given collection. It is assumed that the element is in the collection before the operation is executed.

IS-ELEMENT
: Tests whether a given element is in a given collection.

IS-EMPTY
: Tests whether a given collection has no elements in it.

ANY-ELEMENT
: Given a collection, returns some unspecified element.

Typically, a program uses only a subset of these operations. The particular operations used in a given program strongly influence the utility of certain representations. For some representations, certain operations are impossible (e.g., ANY-ELEMENT when the collection is represented using property list markings). In other cases, the efficiency of an operation can differ significantly for different operations (e.g., IS-ELEMENT for hash table entries and linked lists). Two characteristics of the operations are of primary importance. The first characteristic may be termed *destructive*: the collection operand is physically modified by the operation. ADD-ELEMENT and REMOVE-ELEMENT are both destructive operations. The second characteristic may be termed *element-oriented*: any required elements are known at the time the operation is performed. ADD-ELEMENT, REMOVE-ELEMENT, and IS-ELEMENT are element-oriented; ANY-ELEMENT is not. These considerations will be discussed further in the context of particular rules.

In addition to the basic operations discussed above, there are six operations that deal with several collections simultaneously:

DUPLICATED-COLLECTION
: Creates a new collection whose elements are all elements of another collection.

----

30 Currently PECOS does not differentiate between multisets and sets.

SUBSET
Creates a new collection whose elements are all elements of another collection that satisfy a given predicate.

UNION
Creates a new collection whose elements are all objects that are elements of any collection in an explicit list of collections.

INTERSECTION
Creates a new collection whose elements are all objects that are elements of every collection in an explicit list of collections.

DIFFERENCE
Creates a new collection whose elements are all elements of one collection that are not elements of another.

IS-SUBSET
Tests whether every element of one collection is also an element of another.

Finally, there are four control structures that potentially involve considering all elements of a collection:

FOR-ALL-DO
Performs a given action for every member of a given collection. If a predicate is specified, the action is performed only for the elements satisfying the predicate.

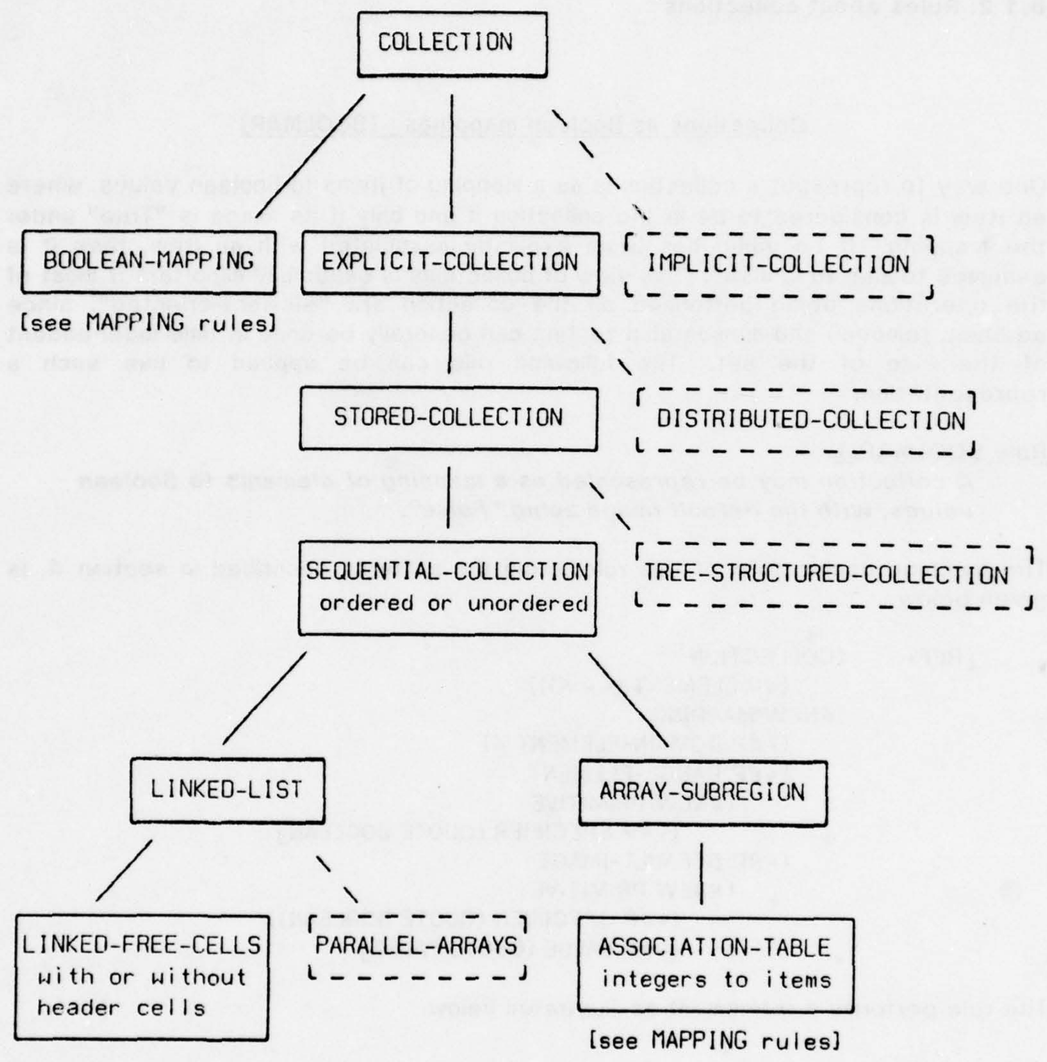
FOR-ANY-DO
Performs a given action for some member of a given collection. The member may be required to satisfy a given predicate. An action to be performed if there is no such element may also be specified.

FOR-ALL-TRUE
Tests whether every element of a given collection satisfies a given predicate.

FOR-ANY-TRUE
Tests whether any element of a given collection satisfies a given predicate.

6.1.1. Overview of collection representations

The following diagram summarizes the representation techniques that PECOS currently employs for collections, as well as several (indicated by dashed lines) that it does not. Each branch represents a refinement relationship. The terms will be defined in the sections dealing with the rules for that representation.


COLLECTION
BOOLEAN-MAPPING
[see MAPPING rules]
EXPLICIT-COLLECTION
IMPLICIT-COLLECTION
STORED-COLLECTION
DISTRIBUTED-COLLECTION
SEQUENTIAL-COLLECTION ordered or unordered
TREE-STRUCTURED-COLLECTION
LINKED-LIST
ARRAY-SUBREGION
LINKED-FREE-CELLS with or without header cells
PARALLEL-ARRAYS
ASSOCIATION-TABLE integers to items
[see MAPPING rules]

6.1.2. Rules about collections

Collections as Boolean mappings (BOOLMAP)

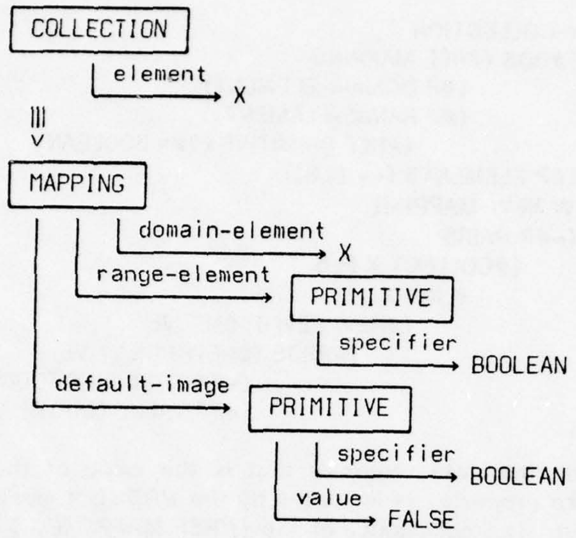
One way to represent a collection is as a mapping of items to Boolean values, where an item is considered to be in the collection if and only if its image is "True" under the mapping. If no value has been explicitly associated with an item, then it is assumed to map to "False". This view of collections is especially important if most of the operations being performed on the collection are "element-oriented", since addition, removal, and membership testing can generally be done in time independent of the size of the set. The following rule can be applied to use such a representation:

Rule BOOLMAP.1:
> ***A collection may be represented as a mapping of elements to Boolean values, with the default image being "False".***

The machine-usable form of this rule, using the patterns described in section 4, is given below:

```
[REF←   (COLLECTION
             (#P ELEMENT (←← X)))
        (#NEW MAPPING
             (←#P DOMAIN-ELEMENT X)
             [←#P RANGE-ELEMENT
                  (#NEW PRIMITIVE
                       (←#P SPECIFIER (QUOTE BOOLEAN]
             (←#P DEFAULT-IMAGE
                  (#NEW PRIMITIVE
                       (←#P SPECIFIER (QUOTE BOOLEAN))
                       (←#P VALUE (QUOTE FALSE]
```

The rule performs a refinement as illustrated below:



Note that the **domain-element** property of the **MAPPING** node is the same as the **element** property of the **COLLECTION** node.

The next rule can be used to implement the operation for creating a collection when the collection is represented as Boolean mapping:

Rule BOOLMAP.2:

> ***A collection represented by a Boolean mapping may be created by creating a mapping; for each initial element (X) of the collection, there is an initial pair <X, "True"> for the mapping.***

The full form of the rule is as shown below[31]:

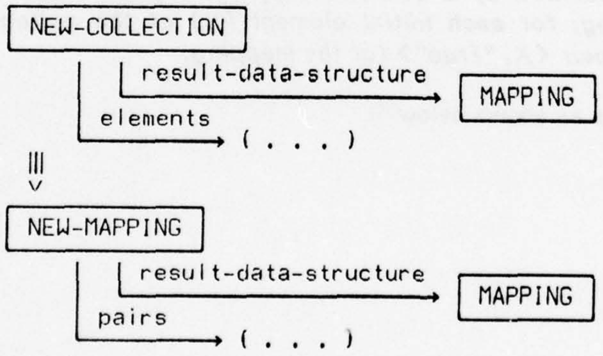
----

[31] The **(#COLLECT ...)** pattern applies a function to every item in a list (in this case the list of initial elements) and returns a list of the results.

```
[REF←     (NEW-COLLECTION
              [#RDS (#REF MAPPING
                          (#P DOMAIN-ELEMENT)
                          (#P RANGE-ELEMENT
                                (#REF PRIMITIVE (?#= BOOLEAN]
              (#P ELEMENTS (←← ELS)))
          (#NEW NEW-MAPPING
              (←#P PAIRS
                   (#COLLECT X ELS
                        (LIST X
                             (#NEW NEW-PRIMITIVE
                                  (←#RDS (#NEW PRIMITIVE
                                              (←#P SPECIFIER (QUOTE BOOLEAN))
                                              (←#P VALUE (QUOTE TRUE]
```

Note the condition on the data structure that is the value of the operation (the **result-data-structure** property, as indicated by the **#RDS**): it must be represented as a Boolean mapping. The complexity of the **(#REF MAPPING ...)** condition on the **result-data-structure** property is related to the idiosyncrasies of the rule formalism; the condition insures that the proper kind of correspondence is being dealt with. An alternative way to handle this would have been to use some construct like **(?QUERY BOOLEAN-CORRESPONDENCE #)**, with the detailed conditions appearing in a rule for dealing with such queries[32].

The rule performs the following refinement:



This rule also illustrates another feature of the rule interpreter: the **result-data-structure** property of a node is automatically inherited by all refinements of the node. No other properties are inherited automatically.

The next rule describes how to refine a test on whether a collection is empty:

----

32 In fact, if I were to write the rules again, I would use such a technique, since it would considerably simplify many of the conditions. See section 11.

Rule BOOLMAP.3:

> *If a collection is represented as a Boolean mapping, a test of whether the collection is empty may be implemented by a test on whether the inverse image of "True" under the mapping is empty.*

The rule's full form is:

```
[REF←    [IS-EMPTY
              (#P COLLECTION (←← C)
                   (#RDS (#REF CORRESPONDENCE
                              (#P RANGE-ELEMENT
                                   (#REF PRIMITIVE
                                        (#P SPECIFIER (?#= BOOLEAN]
          (#NEW HAS-EMPTY-INVERSE
               (←#P CORRESPONDENCE C)
               (←#P RANGE-ELEMENT
                    (#NEW NEW-PRIMITIVE
                         (←#RDS (#NEW PRIMITIVE
                                   (←#P SPECIFIER (QUOTE BOOLEAN))
                                   (←#P VALUE (QUOTE TRUE]
```

Note that the rule conditions are on the **result-data-structure** of the argument operation nodes. As discussed in section 4, the **result-data-structure** property represents the data structure passed from one operation to another, in this case from the operation that produces the collection to the IS-EMPTY operation. Most of the rules have such conditions attached to the **result-data-structures** of their operands.

The membership rule for this representation is similar:

Rule BOOLMAP.4:

> *If a collection is represented as a Boolean mapping, a test of whether an item is in the collection may be implemented as a retrieval of the image of the item under the mapping.*

The next rule considers how to add elements to such collections:

Rule BOOLMAP.5:

> *If a collection is represented as a Boolean mapping, an item may be added to the collection by changing the image of the item from "False" to "True".*

Such a "change image" operation may take different forms, depending on whether the item maps explicitly to "False" or whether it maps to "False" because "False" is the default image. The rules for dealing with such operations will be discussed with the mapping rules in section 6.4. The rule about removing elements also involves changing the image of the item:

Rule BOOLMAP.6:

If a collection is represented as a Boolean mapping, an item may be removed from the collection by changing the image of the item from "True" to "False".

The implementation of an **ANY-ELEMENT** operation involves retrieving some element in the inverse image of "True":

Rule BOOLMAP.7:

The retrieval of some unspecified element of a collection represented as a Boolean mapping may be implemented by a retrieval of some unspecified element of the inverse image of "True" under the mapping.

For some representations of mappings (e.g., property list markings), this operation may not be effectively computable.

A Boolean mapping is a special case of general mappings. The rules for dealing with mappings will be discussed in section 6.4.

Explicit collections (EXPCOL)

The elements of a collection may be represented either explicitly or implicitly. For example, a list of elements is an explicit representation, while upper and lower bounds on a set of integers is an implicit representation. PECOS's rules deal only with explicit representations:

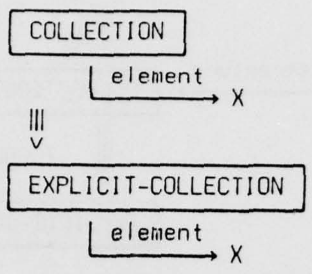
Rule EXPCOL.1:

A collection may be represented explicitly.

The full form of the rule is as follows:

```
(REF←    (COLLECTION
              (#P ELEMENT (←← X)))
         (#NEW EXPLICIT-COLLECTION
              (←#P ELEMENT X)))
```

It performs a refinement as illustrated below:



Note that the **element** properties of the two nodes are the same.

There are two rules dealing with the creation of instances of a collection (the **NEW-COLLECTION** operation), differing on whether or not any elements are specified to be in the collection initially. Typically, collections are created with no elements. The following rule is intended primarily for this case, but is applicable regardless of the initial elements of the collection:

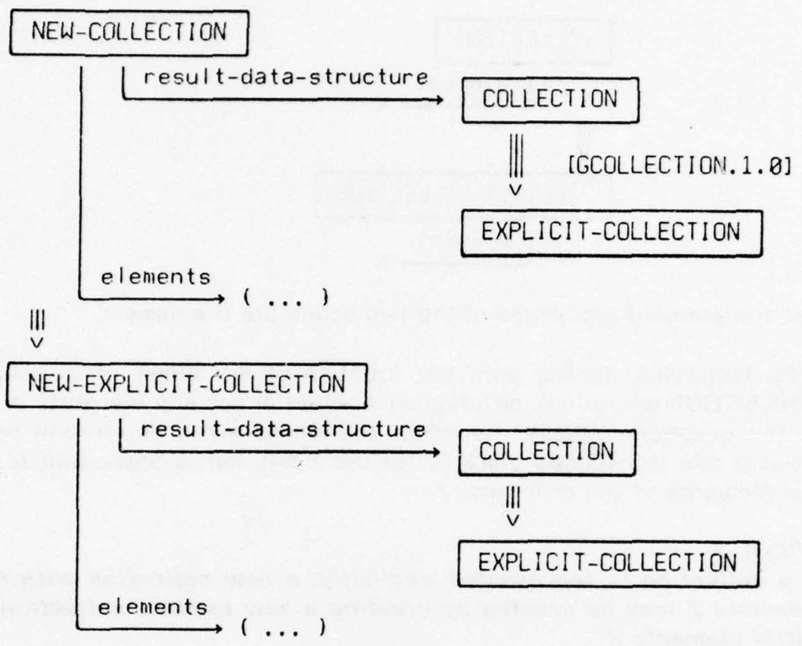
<u>Rule EXPCOL.2</u>:

> ***If a collection is represented explicitly, a new collection with initial elements Z may be created by creating a new explicit collection with initial elements Z.***

This rule is similar to the rule for the case in which the collection is represented as a Boolean mapping. The rule in its full form is as follows:

```
(REF←    (NEW-COLLECTION
             (#RDS (#REF EXPLICIT-COLLECTION))
             (#P ELEMENTS (←← Z)))
         (#NEW NEW-EXPLICIT-COLLECTION
             (←#P ELEMENTS Z)))
```

The condition on the **result-data-structure** property prevents this rule from being applied when the collection is represented as a mapping. The refinement performed by this rule is illustrated below (with the refinement chain for the data structure explicitly indicated):



Note that the **elements** property of the **NEW-COLLECTION** node specifies the initial set of elements. The following rule is applicable only if this set is non-empty:

Rule EXPCOL.3:

> ***A collection with a non-empty list of initial elements may be created by creating a collection with no initial elements and adding each one of the initial elements to this collection.***

One must be careful to distinguish between the case where the list of elements is known at "compile time" and where the list is known at "run time". In PECOS's specification language, the description of the initial elements is an explicit list of operations whose values at run time will be the initial elements, and EXPCOL.3 reflects this. If one wanted to specify a variable list of initial elements, one would have to specify a collection whose elements are to be in the new collection initially.

EXPCOL.3 is actually applicable regardless of the representation of the collection. It is included in the discussion here to illustrate that different refinement rules for the same concept may be applicable in different situations.

As mentioned in section 4.3, for many data structure refinement rules there are corresponding operation refinement rules that are applicable only if the data structure refinement has been made. Rule EXPCOL.2 above is such a rule. In addition, there are similar rules for the ADD-ELEMENT, REMOVE-ELEMENT, IS-ELEMENT, IS-EMPTY, and ANY-ELEMENT operations. In order to focus on the more central issues, such rules will be omitted from this and further discussions.

There are two ways to indicate the elements of a collection explicitly: they can either be stored in a single structure or they can be kept in several structures ("distributed"). For example, using property list markings is one kind of distributed representation. PECOS can currently deal with distributed collections only through the use of distributed mappings; see section 6.4.

The rule for stored collections is the following:

Rule EXPCOL.4:

An explicit collection may be stored in a single structure.

In addition to the above rule, there are six rules for performing the parallel refinements on operations. They will be omitted here.

There are two rules for refining a membership test that are independent of the way that the stored collection is represented:

Rule EXPCOL.5:

A membership test on a stored collection may be refined into test on whether any item in the collection is equal to the item being tested.

The refinement produced by this rule is a FOR-ANY-TRUE test. The rules for dealing with such tests will be discussed in section 6.2.

Notice that there is no requirement that the FOR-ANY-TRUE operation be performed by enumerating the items of the collection, although that is frequently the technique used. The following rule, on the other hand, requires that the elements be enumerated according to a particular ordering relation.

Rule EXPCOL.6:

If there is an ordering relation for the elements of a stored collection, a membership test may be implemented as a total enumeration of the elements of the collection according to the ordering relation; if an element is found that is equal to the item being tested, return "True" as the answer; if an element is found that follows the item being tested, abandon the enumeration; if the enumeration terminates, either through abandonment or through exhaustion of the elements, return "False".

The constraint on the enumeration order is effected by attaching an enumeration-order property to the ENUMERATE-ITEMS node, as illustrated by part of the complete form of EXPCOL.6:

```
[REF←   [IS-STORED-IN-COLLECTION
            [#P COLLECTION (←← Y)
                (#RDS (#REF STORED-COLLECTION
                        (#P ELEMENT (←← XX)
                            (←← OR #ANSWER
                                (?QUERY ORDERING-RELATION #]
            (#P ELEMENT (←← X)
                (#RDS (?QUERY REPRESENTATION-MATCH # XX]
        (#NEW ENUMERATE-ITEMS
            . . .
            (←#P ENUMERATION-ORDER (LIST (QUOTE ORDERED) OR))
            . . .]
```

The role of the **enumeration-order** property will be discussed in connection with the rules for enumeration structures in section 6.2.

<u>Collections grouped in sequential structures (SEQCOL)</u>

Several types of structures can be used for storing the elements of a collection. PECOS's rules only cover one technique: the use of a sequential collection.

<u>Rule SEQCOL.1</u>:

A stored collection may be represented using a sequential collection.

A sequential collection may be thought of as a linear arrangement of locations, where each location contains a single element of the collection.

The rules for dealing with the "any element" operation illustrate the relationship between locations in sequential collections and the elements that are stored in the locations:

<u>Rule SEQCOL.2</u>:

The retrieval of any unspecified element from a collection represented as a sequential collection may be implemented by the retrieval of the element at any unspecified location.

The full form of this rule is given below:

```
(REF←   [ANY-STORED-ELEMENT
            (#RDS (←← X))
            (#P COLLECTION (←← C)
                (#RDS (#REF SEQUENTIAL-COLLECTION
                        (#P ELEMENT
                            (?QUERY REPRESENTATION-MATCH # X]
        (#NEW ELEMENT-AT-LOCATION
            (←#P COLLECTION C)))
```

An ELEMENT-AT-LOCATION node normally has both a **collection** and a **location** property. Note that SEQCOL.2 specifies only the **collection** property. The fact that no **location** property is specified reflects the "any unspecified location" of the English version.

The retrieval of an element at a location in a collection can't be implemented until the location is determined. In some situations this may involve searching through the collection. In this particular case, any location at all may be used. PECOS currently has only one rule for determining an unspecified location for an **ELEMENT-AT-LOCATION** operation:

Rule SEQCOL.3:

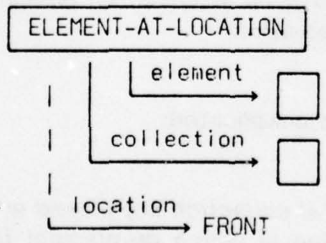
The retrieval of the element at any unspecified location may be specified to retrieve the element at the front.

This rule simply specifies one of the many possible locations that could be used. For sequential collections, the front and the back are often convenient, since these are generally more easily isolated than, for example, the "middle" location. In fact, the front and back differ in their utility for different implementations of sequential collections. In particular, for arrays front and back are generally equally convenient since the upper and lower bounds of the array are known (but see the discussion on growing, shrinking, and fixed boundaries in section 6.1.4). For linked lists, the front is generally more accessible than the back.

The full form of the rule illustrates the use of property rules:

```
(PROP←  LOCATION
        (ELEMENT-AT-LOCATION)
        (QUOTE FRONT))
```

When applied, this rule adds the **location** property to an **ELEMENT-AT-LOCATION** node, as illustrated below:



Thus, the rule further specifies a previously unspecified aspect of a node. Property rules are used quite frequently for such purposes.

Removing an element from a sequential collection is slightly more complicated, as shown by the following rule:

Rule SEQCOL.4:

If a stored collection is represented as a sequential collection, an element may be removed from the collection by finding the location of the element and removing the item at that location.

The addition of an element to a collection involves the notion of a position in such a collection. A POSITION may be thought of as the space "between" two locations. Note that a collection with *n* elements has *n* locations, but *n+1* positions (including the positions before the first element and after the last). The following rule makes use of such a notion:

Rule SEQCOL.5:

If a stored collection is represented as a sequential collection, an element may be added to the collection by adding it at an unspecified position.

The position is unspecified and must be determined. The determination of the position is complicated by the fact that the elements of a sequential collection may be stored according to some ordering relation. When the collection is unordered, elements may be added at any convenient position; when the collection is ordered, the precise position must usually be found by searching for it. In the unordered case, the two possibilities are given by the following rules:

Rule SEQCOL.6:

If the elements of a sequential collection are not stored according to any ordering relation, the position at which an element is added may be specified to be the front position.

Rule SEQCOL.7:

If the elements of a sequential collection are not stored according to any ordering relation, the position at which an element is added may be specified to be the back position.

The ordered case is somewhat more complicated:

Rule SEQCOL.8:

If the elements of a sequential collection are stored according to some ordering relation, the position at which an element is added may be specified to be the result of an operation of finding the position of the element in the collection.

There are several ways that such positions may be represented. Among the more common are pairs of location indicators (giving the two locations between which the position occurs) and a single indicator giving either the preceding location or the following location. PECOS's rules deal only with the last of these:

Rule SEQCOL.9:

> ***A position for inserting an element in a sequential collection may be represented as the location which immediately follows the position.***

Once this method of representing a position between two locations has been selected, the following rule can be applied to implement the actual addition operation:

Rule SEQCOL.10:

> ***If an insertion position is represented as the location that follows the position of the new element, the element may be added by inserting it before the location.***

The rules for inserting an element before a location depend on the representation of the sequential collection and will be covered later.

6.1.2.1. Summary for sequential collections

Before considering techniques for implementing sequential collections, it may be helpful to summarize the data structures and operations at this refinement level. There are two principal data structures:

SEQUENTIAL-COLLECTION
A linearly related set of locations that contain the elements of the collection.

LOCATION-IN-COLLECTION
A way of referring to locations in a sequential collection.

There are seven operations:

NEW-SEQUENTIAL-COLLECTION
Creates a new sequential collection.

TEST-EMPTY-SEQUENTIAL-COLLECTION
Tests whether a sequential collection is empty.

IS-STORED-IN-SOME-LOCATION
Tests whether an item is stored in any location of a sequential collection.

ELEMENT-AT-LOCATION
Returns the element at a particular location in a sequential collection.

INSERT-ELEMENT-AT-POSITION
Inserts an element at a position in a sequential collection when the position is specified as **FRONT** or **BACK**.

INSERT-BEFORE-LOCATION
Inserts an item before a location in a sequential collection when the location is specified as a **LOCATION-IN-COLLECTION**.

REMOVE-ELEMENT-AT-LOCATION
Removes the element at a location in a sequential collection when the location is specified as a **LOCATION-IN-COLLECTION**.

6.1.3. Rules about linked lists

One way to implement a sequential collection is to use a linked list: each location contains one element and a link indicating the next location in the sequence[33]. The principal features of linked lists are that insertions, deletions, and rearrangements are relatively easy: the links can be manipulated without changing the items stored in the locations. The price paid for this flexibility is that a list can be accessed in only one direction, from the first element to the last element. One consequence is that the time savings that often result from keeping a sequential collection ordered are not as significant with lists. For example, binary searching methods do not apply to linked lists.

Sequential collections as linked lists (LIST)

The data structure refinement rule is similar to those already considered:

Rule LIST.1:

A sequential collection may be represented as a linked list.

There are many ways that linked lists can be implemented. For example, when the data are record structures, there can be a field in the structure that contains an indicator of the next datum. Parallel arrays can also be used: one array contains the items and one array contains the links (indices). A third way involves the allocation of cells from free storage. Necessary ingredients for such an implementation include some way of allocating such cells and some way of retrieving cells no longer in use (garbage collection). Each cell is considered to have two parts, an **item** (the element stored in the cell) and a **link** (a pointer to the next cell). A special flag is generally used as an indication of an empty list or as the list terminator (the link of the last cell). Typically, the same flag will be used for both purposes, since it simplifies many algorithms (especially recursive algorithms that trace down the links of a list).

PECOS's rules deal with some aspects of the use of free cells to implement linked lists. They assume that allocation and garbage collection are handled automatically and that the same flag is used both to terminate lists and to indicate the empty list.

The first rule is the data structure refinement rule:

Rule LIST.2:

A linked list may be represented using linked free cells.

When dealing with linked free cells, it is often helpful to use a special cell at the head of such a list. In such cases, the list is always a pointer to a particular cell.

----

33 Multiply-linked lists are a variation not covered by the rules.

Without such a header cell, a list may be a pointer to a cell or it may be the empty list flag (e.g., either a CONS cell or NIL in LISP). Using a header cell facilitates addition and removal of elements, since the special case of the empty list can easily be ignored. PECOS has rules for lists with and without such special header cells. The following two rules are used to select one of these possibilities:

Rule LIST.3:
: *Linked free cells may be used without a special header cell.*

Rule LIST.4:
: *Linked free cells may be used with a special header cell which is a primitive with value "HEAD"*[34].

These rules are both implemented using *property* rules. For example, the internal form of LIST.3 is given below:

```
(PROP← SPECIAL-HEADER-CELL (LINKED-FREE-CELLS)
       NIL)
```

Property rules such as these are often used to deal with minor variations of a basic notion.

The implementations of operations on linked free cells generally differ only slightly in the two cases. The two rules for creating a new instance of a linked list are as follows:

Rule LIST.5:
: *If a linked list is represented as linked free cells without a special header cell, a new list may be created by creating a new instance of the empty link flag.*

(Recall that the same flag is used for both the empty list and for the list terminator.)

Rule LIST.6:
: *If a linked list is represented as linked free cells with a special header cell, a new list may be created by allocating a new cell whose item part is a new instance of the special header and whose link part is a new instance of the empty link flag.*

Testing whether a list is empty is also different in the two cases:

----

[34] The "HEAD" is simply a tag denoting the header cell. There is, of course, no necessity that any particular value be used. In fact, having the value be a link to some distinguished cell in the list may facilitate other operations. For example, if the header contains a pointer to the last cell in the list, the insertion of elements at the back of the list is relatively easy. PECOS currently has no rules for such special purpose cells.

Rule LIST.7:

If a linked list is represented with linked free cells without a special header cell, a test of whether the list is empty may be implemented by a test of whether the list is the empty link flag.

Rule LIST.8:

If a linked list is represented with linked free cells with a special header cell, a test of whether the list is empty may be implemented by a test of whether the link of the first cell of the list is the empty link flag.

The only difference between the two is that with a special header cell, the link from the first cell must be taken before the test can be applied. The same holds true for a membership test:

Rule LIST.9:

If a linked list is represented with linked free cells without a special header cell, a test of whether an item is in the list may be implemented by a test of whether the item is in one of the cells of the list.

Rule LIST.10:

If a linked list is represented with linked free cells with a special header cell, a test of whether an item is in the list may be implemented by a test of whether the item is in one of the cells of the list pointed to by the link of the first cell.

Finally, the retrieval of the element at the front of a list is dependent on whether or not there is a header cell:

Rule LIST.11:

If a linked list is represented as linked free cells without a special header cell, the retrieval of the element at the front position of the list may be implemented by a retrieval of the item part of the first cell of the list.

Rule LIST.12:

If a linked list is represented as linked free cells with a special header cell, the retrieval of the element at the front position of the list may be implemented by a retrieval of the item part of the cell pointed to by the link of the first cell in the list.

With destructive operations applied to linked free cells without a header, the situation is complicated by the fact that the empty list must be dealt with as a special case. When adding an element to an empty list, the list is changed from an empty list flag to a free cell. Usually, this cannot be done by simply manipulating pointers. It is necessary to determine the original "source" of the structure (e.g., the variable whose value is the collection), so that this source may be modified to have a cell as its value. Likewise, when removing the only element of a list with no header, the result is the empty link flag. Again, this cannot be achieved by simply manipulating pointers, and the source of the collection must be known. In general, determining the source of a structure can be rather difficult, especially in languages whose variables may assume pointer values (such as LISP), and PECOS has no rules for dealing with this case. The effect is that PECOS cannot implement any destructive operations on linked lists without header cells. The rules for destructive operations on lists with header cells will be considered after a brief discussion of the notion of a location in a linked list.

Locations in linked lists (LISTLOC)

A location in a linked list is some indication of a particular cell of the list. There are several ways to indicate cells in linked lists. PECOS's rules deal with two of them[35]:

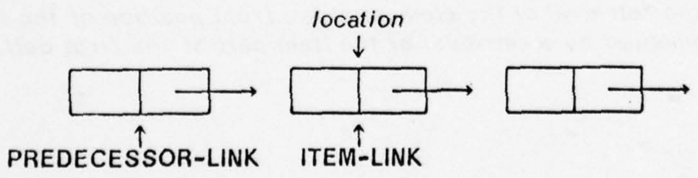
Rule LISTLOC.1:

If a linked list is represented as linked free cells, a location may be represented as a link to the cell of the location.

Rule LISTLOC.2:

If a linked list is represented as linked free cells with a special header cell, a location may be represented as a link to the cell preceding the location.

An illustration may help to clarify the difference between these two location representations:



The retrieval of the item at a location in a linked list is dependent on the location representation, as seen in the following two rules:

----

[35] Although the rules are stated in terms of linked free cells, the notions involved also apply to other representations for linked lists.

Rule LISTLOC.3:

If a location is represented as a link to the cell of the location, a retrieval of the item at the position may be implemented by retrieving the item part of the cell indicated by the location representation.

Rule LISTLOC.4:

If a location is represented as a link to the cell preceding the cell of the location, a retrieval of the item at the location may be implemented by retrieving the item part of the cell pointed to by the link of the cell indicated by the location representation.

Destructive operations on linked free cells with headers (LISTDEST)

PECOS's rules for destructive operations on linked lists deal only with linked free cells with headers. For inserting an element, the simplest case is when the position is specified to be the front of the list:

Rule LISTDEST.1:

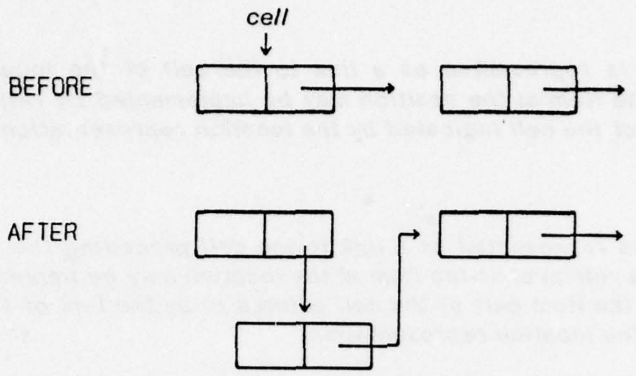
If a linked list is represented as linked free cells with a special header cell the insertion of an element at the front position may be implemented as an insertion of the element after the first cell of the list.

The "insertion" operation can be implemented by applying the following rule:

Rule LISTDEST.2:

An insertion of an item after a cell in a linked list may be implemented by replacing the link of the cell by a pointer to a new cell whose item part is the new item and whose link part is the link part of the original cell.

An illustration may clarify this operation:



The other insertion operation for sequential collections involves inserting an element before a given location. PECOS can only deal with the case where the location is represented as a predecessor link. The difficulties of dealing with item links are similar to those of dealing with lists without header cells.

Rule LISTDEST.3:

> ***If a location is represented as a link to the cell preceding the location, an insertion of an element before the location may be implemented as an insertion of the element after the cell indicated by the location representation.***

Since the cell indicated by the location is the cell after which the element is to be inserted, the cell whose link must be modified is accessible. This is the primary virtue of using header cells and predecessor links to represent locations. Once the above rule has been applied, the previously given rule (LISTDEST.2) for inserting an item after a cell may be used.

The rules for removing an element at a location in a linked list are similar to those for inserting an element. The simplest case is again where the location is at the front:

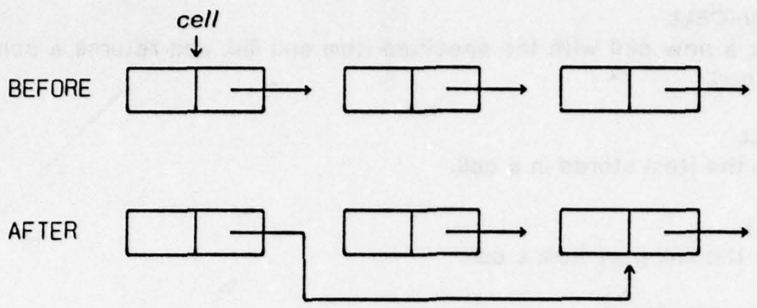
Rule LISTDEST.4:

> ***If a linked list is represented as linked free cells with a special header cell the removal of an element at the front location may be implemented as a removal of the cell after the first cell of the list.***

Rule LISTDEST.5:

> ***A removal of the cell after a cell in a linked list may be implemented by replacing the link of the cell by the link part of the cell pointed to by the link part of the original cell.***

An illustration may also be helpful here:



The removal of an element at a location can also only be done if the location is represented as a predecessor link:

Rule LISTDEST.6:

> ***If a location is represented as a link to the cell preceding the location, the removal of the element at the location may be implemented the removal of the cell after the cell indicated by the location representation.***

6.1.3.1. Summary for linked free cells

The following data structures and operations constitute those required at this level of refinement:

LINKED-FREE-CELLS
: Each cell contains an item and a link.

ITEM-LINK
: A link to the cell of a particular item (or the list terminator flag).

PREDECESSOR-LINK
: A link to the cell preceding that of a particular item.

NEW-EMPTY-FREE-LINK
: Creates a new instance of the empty list flag.

TEST-EMPTY-FREE-LINK
: Tests whether a link is the empty list flag.

IS-STORED-IN-SOME-LINKED-FREE-CELL
: Tests whether an element is stored in any free cell in a linked list.

LINK-TO-NEW-CELL
Creates a new cell with the specified item and link and returns a pointer to that cell.

ITEM-OF-CELL
Returns the item stored in a cell.

LINK-OF-CELL
Returns the link part from a cell.

REPLACE-LINK-OF-CELL
Replaces the link of a cell with another link.

6.1.4. Rules about contiguous regions of arrays

Another standard technique for representing a sequential-collection is to use a contiguous subregion of an array, where the linear relationship between locations is that defined by the indices of the array. Every location in the contiguous subregion contains one of the elements of the collection. Such a representation will be termed an "array subregion". Conceptually, such structures actually consist of three parts: a lower bound, an upper bound, and an allocation of consecutive cells from storage. Typically, the bounds are changed dynamically as elements are added and removed, while the allocation remains fixed. The lower bound will be considered to be the index of the first element (the front) and the upper bound will be considered to be the index of the last element (the back). There are two principal virtues of array subregions as collection representations. The primary advantage comes with the use of ordered collections: searching for a position or element can be done fairly quickly using binary search techniques. On the other hand, insertions and deletions on such ordered collections can be relatively expensive since shifts are required. A less significant advantage is that less space is required than for linked lists, since no memory is needed to store the links to each location's successor. In addition, no facilities for free storage allocation or garbage collection are required.

Although the discussion of array subregions will use such terms as array and index, the notion is actually somewhat more general than that. All that is required of the "indices" is that there be some way of incrementing and decrementing them. All that is required of the "storage allocation" is that there be some way of determining the element that corresponds to a particular "index". Any way of representing this mapping of integers to elements would suffice. For example, a list of <integer, element> pairs could be used. For the sake of clarity, the intermediate step between sequential collections and the standard array representation will be ignored in the following discussion.

<u>Sequential collections as array subregions (ARRAY)</u>

The data structure refinement rule is similar to that for linked lists:

<u>Rule ARRAY.1</u>:

> ***A sequential collection may be represented as an array subregion with an allocation, a lower bound, and an upper bound.***

When adding an element to an array subregions, it is necessary to expand the region so that a location may be made available for the new element. Similarly, when removing an element the region must be shrunk so there will be no locations that do not contain elements. PECOS's rules deal only with the case in which a specific boundary is identified as being the "growing" boundary, and a specific boundary is identified as the "shrinking" boundary. The case in which the growing boundary and shrinking boundary are selected dynamically is not covered. In fact, the current set of rules only deals with the case in which the growing and shrinking boundaries are the same and in which a particular boundary is identified as a "fixed" boundary. The following rules permit the selection of particular boundaries for these purposes.

In some situations PECOS is free to select the boundary to be used as the fixed boundary. The first rule allows this selection be made:

Rule ARRAY.2:
If the fixed boundary of an array subregion is unspecified, the lower bound may be used.

There is no particular reason that the lower bound should be preferred over the upper bound. In fact, many interesting cases arise when the same allocation is shared by two separate array subregions, with the two growing in opposite directions. For example, many iterative sorting programs fit this paradigm [Green and Barstow 1977b]. The remaining four rules permit PECOS to deal with array subregions in which either boundary is specified to be the fixed boundary:

Rule ARRAY.3:
If the fixed boundary of an array subregion is specified to be the lower bound, the growing boundary may be specified to be the upper bound.

Rule ARRAY.4:
If the fixed boundary of an array subregion is specified to be the lower bound, the shrinking boundary may be specified to be the upper bound.

Rule ARRAY.5:
If the fixed boundary of an array subregion is specified to be the upper bound, the growing boundary may be specified to be the lower bound.

Rule ARRAY.6:
If the fixed boundary of an array subregion is specified to be the upper bound, the shrinking boundary may be specified to be the lower bound.

Many of the rules for dealing with array subregions must differentiate between the two cases. There are two rules for creating a new array subregion:

Rule ARRAY.7:

A sequential collection represented as an array subregion with fixed lower bound may be created by creating a new allocation, a new lower bound which is the minimum value of the index and an upper bound which is one less than the minimum value of the index.

Rule ARRAY.8:

A sequential collection represented as an array subregion with fixed upper bound may be created by creating a new allocation, a new lower bound which is one less than the maximum value of the index and an upper bound which is the maximum value of the index.

Note the dependence of the index values on the range of possible values for the index. Generally it is necessary to determine both the size of the array and the range of index values. These depend on such parameters as the expected and maximum size of the collection under consideration. Note also that the initial values of the upper and lower bounds depend on which boundary is fixed. The rule for testing whether an array subregion is empty, however, applies in both cases:

Rule ARRAY.9:

A test of whether a sequential collection represented as an array subregion is empty may be implemented by a test of whether the lower bound is greater than the upper bound.

Regardless of which boundary is fixed, if the lower bound is greater than the upper bound, the subregion is empty.

PECOS has no rules for refining membership tests at the level of array subregions. Membership tests for such collections are constructed through the application of either rule EXPCOL.5 or rule EXPCOL.6, discussed in section 6.1.2 earlier.

The retrieval of the element at either the front or back is straightforward:

Rule ARRAY.10:

If a sequential collection is represented as an array subregion, the retrieval of the element at the front location may be implemented by retrieving from the allocation the item stored at the lower bound.

Rule ARRAY.11:

If a sequential collection is represented as an array subregion, the retrieval of the element at the back location may be implemented by retrieving from the allocation the item stored at the upper bound.

Locations in array subregions (ARRAYLOC)

There are many ways to indicate locations in array subregions. The most natural is to simply use the index of the location, but the index of the preceding or following location is also possible. PECOS's rules only deal with the use of the index of the location:

Rule ARRAYLOC.1:

If a sequential collection is represented as an array subregion, a location may be represented as an index.

The corresponding rule for an **ELEMENT-AT-LOCATION** operation is:

Rule ARRAYLOC.2:

If a sequential collection is represented as an array subregion, the retrieval of the element at a location indicated by an index may be implemented by retrieving from the allocation the item stored at that index.

Inserting an element to an array subregion (ARRAYINS)

In general, adding an element will require that the subregion be expanded, and that the old elements be shifted to make room for the new one[36]. Two of the rules for inserting an element at a particular position avoid the shift by taking advantage of the fact that the element is being inserted at the growing boundary:

Rule ARRAYINS.1:

If the growing boundary of an array subregion is the upper bound, the insertion of an element at the back may be implemented by expanding the array subregion by 1 location and depositing the element at the location indicated by the new upper bound.

Rule ARRAYINS.2:

If the growing boundary of an array subregion is the lower bound, the insertion of an element at the front may be implemented by expanding the array subregion by 1 location and depositing the element at the location indicated by the new lower bound.

Inserting an element at an end which is not the growing boundary is somewhat more complicated:

----

[36] In addition, the expanded bounds must be checked to insure that they are not outside the limits of the allocation. PECOS's rules do not include this aspect of array manipulation.

Rule ARRAYINS.3:

> ***If the growing boundary of an array subregion is the lower bound, the insertion of an element at the back may be implemented by expanding the array allocation by 1, shifting all of the elements down by 1, and depositing the new element in the location indicated by the upper bound.***

Rule ARRAYINS.4:

> ***If the growing boundary of an array subregion is the upper bound, the insertion of an element at the front may be implemented by expanding the array allocation by 1, shifting all of the elements up by 1, and depositing the new element in the location indicated by the lower bound.***

The "expansion" referred to in the above rules merely expands the bounds without doing any shifting, addition, or removal or any elements. Such an operation can be implemented through the use of the following two rules.

Rule ARRAYINS.5:

> ***If the growing boundary of an array subregion is the upper bound, the array may be expanded by incrementing the upper bound by 1.***

Rule ARRAYINS.6:

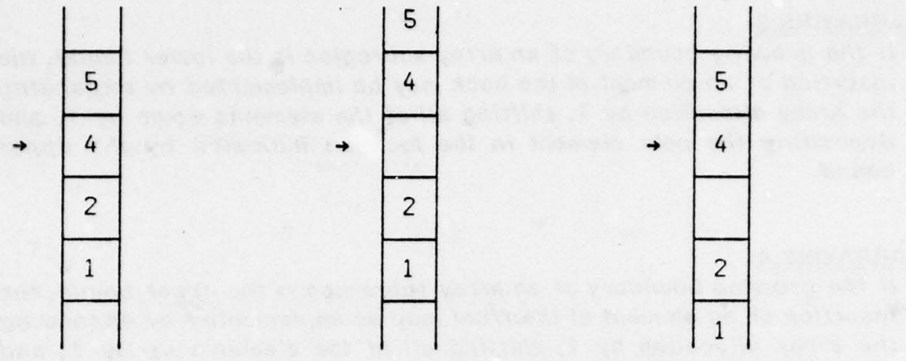
> ***If the growing boundary of an array subregion is the lower bound, the array may be expanded by decrementing the lower bound by 1.***

Finally, the general case of inserting an element before an arbitrary location also involves some shifting.

Rule ARRAYINS.7:

> ***If a sequential collection is represented as an array subregion, an element may be inserted before a location by expanding the allocation by 1, by vacating the position before the location, and by inserting the element into the vacated position.***

The operation of vacating the position before a location will insure that the position is not occupied, so that the element can be deposited without overwriting any other elements. Although this ARRAYINS.7 is applicable for both growth directions, vacating the position before the location differs for the two cases. Before presenting the rules, it may be helpful to illustrate the operation. The figure below shows the situation before vacating the position before the "4" (so that, say, "3" could be inserted between "2" and "4"). Also shown are the situations after the "vacate" has been performed in the upward and downward directions.



item index after vacating upward after vacating downward

Note that in the "upward" case the index specifies the physical location that was vacated, while in the "downward" case it specifies the physical location after that vacated.

Rule ARRAYINS.8:

> ***If the growing boundary of an array subregion is the upper bound, the position before a location indicated by an index may be vacated by shifting the elements from the index through the upper bound up by 1.***

Rule ARRAYINS.9:

> ***If the growing boundary of an array subregion is the lower bound, the position before a location indicated by an index may be vacated by shifting the elements from the lower bound through the 1 less than the index down by 1.***

The two rules for storing an item before a subregion position display the same asymmetry:

Rule ARRAYINS.10:

> ***If the growing boundary of an array subregion is the upper bound, an element may be stored before a vacated position (indicated by an index) by depositing the element in the allocation at the index.***

Rule ARRAYINS.11:

> ***If the growing boundary of an array subregion is the lower bound, an element may be stored before a vacated position (indicated by an index) by depositing the element in the allocation of 1 less than the index.***

Removing an element from an array subregion (ARRAYREM)

The considerations involved in removing an element at a location in an array subregion are similar to those for inserting an element, except that the appropriate part of the subregion must be shrunk rather than expanded:

Rule ARRAYREM.1:

If the shrinking boundary of an array subregion is the upper bound, an item may be removed from a location by shifting the elements from the index after the location through the upper bound down by 1 and shrinking the array subregion.

Rule ARRAYREM.2:

If the shrinking boundary of an array subregion is the lower bound, an item may be removed from a location by shifting the elements from lower bound through the index before the location up by 1 and shrinking the array subregion.

Shrinking an array is quite similar to expanding an array: only the bounds are affected explicitly.

Rule ARRAYREM.3:

If the shrinking boundary of an array subregion is the upper bound, the array may be shrunk by decrementing the upper bound by 1.

Rule ARRAYREM.4:

If the shrinking boundary of an array subregion is the lower bound, the array may be shrunk by incrementing the lower bound by 1.

Shifting an array subregion (ARRAYSHIFT)

Several of the rules for vacating positions and removing elements involved shifting a part of the array subregion. Such an operation may be viewed in many ways, including the enumeration of positions, a transfer between two collections, and a sequence of operations changing the correspondent of successive indices [Green and Barstow 1977b]. Critical to all of these is an understanding that the shift must be performed in such a way that no elements are overwritten and lost. PECOS has two rules for shifting, one for each direction, but they are probably short-cut rules in that some of the reasoning process is hidden.

Rule ARRAYSHIFT.1:

A part of an array subregion may be shifted down by enumerating the locations between the lower and upper bound in the stored order; for each location, the item at that location is deposited into the location whose index is the location minus the amount of the shift.

Rule ARRAYSHIFT.2:

A part of an array subregion may be shifted up by enumerating the locations between the upper and lower bound in the reverse of the stored order; for each location, the item at that location is deposited into the location whose index is the location plus the amount of the shift.

6.1.4.1. Summary for array subregions as association tables

The following data structures and operations constitute those required at this level of refinement:

ARRAY
: Some way of associating a distinct item with every index (integer) in a given range.

INDEX
: For our purposes, always an integer in a given range.

PLEX
: Since an array subregion consists of three parts (allocation, lower bound, upper bound), a facility for dealing with structures with multiple parts is needed. The techniques uses often depend on the target language, and whether or not that language has facilities for dealing with record structures or array pointers.

NEW-ARRAY
: A way to allocate a new block of storage for an array. The techniques for handling this operation also depend on the target language. In particular, in some languages arrays can be allocated only at compile time or at certain specified times during execution. In INTERLISP, array pointers can be passed as variable values.

DEPOSIT-IN-ARRAY
: A way to deposit an item with a given index in a given array.

RETRIEVE-FROM-ARRAY
: A way to retrieve the item associated with a given index in a given array.

In addition, techniques for enumerating, incrementing, and decrementing indices are needed.

6.1.5. Rules about other collection operations

Of the top-level collection operations mentioned earlier, six deal with several collections simultaneously:

DUPLICATED-COLLECTION
: Creates a new collection whose elements are all elements of another collection.

SUBSET
: Creates a new collection whose elements are all elements of another collection that satisfy a given predicate.

UNION
: Creates a new collection whose elements are all objects that are elements of any collection in an explicit list of collections.

INTERSECTION
: Creates a new collection whose elements are all objects that are elements of every collection in an explicit list of collections.

DIFFERENCE
: Creates a new collection whose elements are all elements of one collection that are not elements of another.

IS-SUBSET
: Tests whether every element of one collection is also an element of another.

PECOS can currently deal with only three of these (DUPLICATED-COLLECTION, SUBSET, IS-SUBSET). The knowledge needed for the other three remains to be codified.

Other collection operations (COLMISC)

The idea of duplicating a collection is that, given a collection with certain elements, another collection with the same elements should be created. At the abstract level of "collection" the two data structures involved are of the same type; hence, the term "duplicated". However, at more refined levels, it is perfectly possible for the two collections to be represented differently. In fact, a frequent use of this operation is to convert from one representation to another. The rule for duplicating collections is as follows:

Rule COLMISC.1:

A collection may be duplicated by a sequence of two actions: first, initialize the new collection with no elements; then, for each element of the original collection, add the element to the new collection.

Note that there must be some way to perform an action for all elements of the original collection. For certain representations, this may not be feasible. For example, if a collection is represented using LISP property list markings, every atom's property list would have to be examined, a very time consuming task. Of course, this problem only arises in connection with the original collection, and not with the duplicated one. For example, it is quite simple to start with a linked list and produce a property list marking.

The rules for computing and testing subsets are also fairly simple:

Rule COLMISC.2:

The subset S of a collection C, such that every element of S satisfies a predicate P, may be computed by a sequence of two actions: first, initialize S with no elements; then, for each element of C, if the element satisfies P add it to S.

Rule COLMISC.3:

A test of whether a collection C_1 is a subset of a collection C_2 may be implemented as a test of whether all elements of C_1 are members of C_2.

PECOS can deal with the union operation in one specific case:

Rule COLMISC.4:

If the "collection" operand of a membership test is computed by computing the union of several collections, the test may be implemented as an "or" of membership tests on each of the collections.

This is one of the few rules whose conditions apply to the operation that produces one of the operands, rather than to the data structure produced by that operation. Such operation simplifications are fairly common in optimizing transformations **[Standish et al 1976]**.

6.2. Enumerations over collections

As mentioned earlier, there are four control structures that may involve considering each of the elements of a collection:

FOR-ALL-DO
: Performs a given action for every member of a given collection. If a predicate is specified, the action is performed only for the elements satisfying the predicate.

FOR-ANY-DO
: Performs a given action for some member of a given collection. The member may be required to satisfy a given predicate. An action to be performed if there is no such element may also be specified.

FOR-ALL-TRUE
: Tests whether every element of a given collection satisfies a given predicate.

FOR-ANY-TRUE
: Tests whether any element of a given collection satisfies a given predicate.

<u>High-level enumeration operations (FOR)</u>

In a FOR-ALL-DO construct, there is no necessity that the elements be considered sequentially (i.e., one after another). For example, with languages or machines that support parallel processes, a separate process could be started for each element of the collection. However, PECOS's rules deal only with the sequential case:

<u>Rule FOR.1</u>:
The process of performing an action A for all elements of a collection may be implemented by a total enumeration of the elements; if a predicate is specified, the action for each element consists of testing the predicate and performing A if the test succeeds; if no predicate is specified, the action for each element is A*[37]*.

The notion of enumerating the items in a collection is central to most of the rules in this section, and will be elaborated in more detail after discussions of the rest of the top-level constructs.

With FOR-ANY-DO constructs, it is often necessary to distinguish between two

----

[37] PECOS's rule actually includes somewhat more detail, in that initial and final actions (for the FOR-ALL-DO) may also be specified, and these are passed on to the **ENUMERATE-ITEMS** construct created by an application of FOR.1.

cases: (a) the collection is empty (or no elements satisfy the predicate); (b) there is an element that satisfies the predicate. For this reason, **FOR-ANY-DO** constructs may also specify an action to be performed if no satisfactory element is found. In the discussions below, this action will be referred to as the "not-found" action.

There are two rules dealing with **FOR-ANY-DO** constructs. The first is for the case in which no predicate is specified, so there is no need to perform any kind of enumeration to try to find an element satisfying the predicate:

Rule FOR.2:

> ***If no predicate is specified, the process of performing an action A for any element of a collection may be implemented as a test of whether the collection is empty; if the test succeeds, the "not-found" action is executed; if the test fails, A is performed on the result of retrieving any element of the collection[38].***

If a predicate is specified, then some element satisfying that predicate must be found. The next rule enables this to be done by searching for such an element[39].

Rule FOR.3:

> ***The process of performing an action A for any element of a collection such that the element satisfies a predicate P may be implemented as a search in the collection for an item satisfying P; if found, A is performed; if not, the "not-found" action is performed.***

There are many ways that such a search can be implemented. If the results of testing the predicate on one element can be used to guide the process, relatively complex search strategies can be developed. The only case covered by PECOS's rules, however, is simply to enumerate the items, one after another, testing each in turn:

Rule FOR.4:

> ***A search in a collection for an element satisfying a predicate P may be implemented as a total enumeration of the items in the collection; the action for each item is a test of whether the item satisfies P; if so, the enumeration halts and the "found" action is performed; if all elements are enumerated (and none satisfies P), the "not found" action is performed.***

Note that the **FOR-ALL-DO** and **FOR-ANY-DO** constructs have both been refined into **ENUMERATE-ITEMS** constructs. The same is true for the two predicates,

----

38 The "retrieve any element" refers to the **ANY-ELEMENT** operation discussed with the collection rules in section 6.1.

39 Again, PECOS's rules do not cover any non-sequential ways of finding the element.

FOR-ANY-TRUE and **FOR-ALL-TRUE**. The **FOR-ANY-TRUE** is first refined into a search:

<u>Rule FOR.5</u>:

> ***A test of whether any element in a collection satisfies a predicate P may be implemented as a search in the collection for an element satisfying P; if such an item is found, return "True"; if not, return "False".***

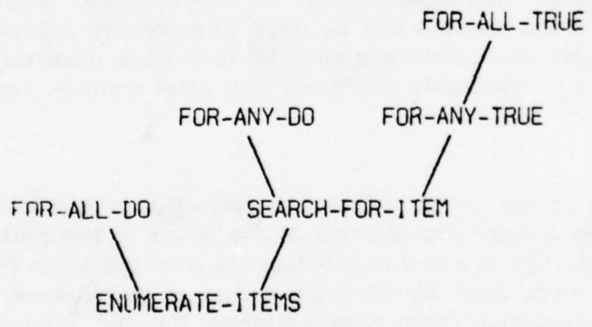
The search rule given earlier (FOR.4) may now be used to refine this into a total enumeration.

The most common way of testing whether every element in a collection satisfies a predicate is to determine whether any element fails to satisfy the predicate. This technique is embodied in the following rule:

<u>Rule FOR.6</u>:

> ***A test of whether all elements of a collection satisfy a predicate P may be implemented as the negation of a test of whether any element of the collection satisfies the negation of P.***

At this point, the previously given rule can be applied, and eventually a total enumeration over the collection is reached. Thus, all four of the top-level constructs are refined into a single notion, that of enumerating the items in a collection. The refinement relationships between these constructs are summarized below:



6.2.1. Enumerating the items in a collection

In its most general form, enumerating the items in a collection can be viewed as an independent process or coroutine. Each call produces one item from the collection. The process must guarantee that every item will be produced on some call and that each will be produced only once. In addition, there must be some way to indicate that all of the items have been produced, as well as some way to start up the process initially. It may also be useful to constrain the process to produce the items in a particular order. For example, EXPCOL.6, a membership test rule given earlier, requires that the items be enumerated according to an ordering relation. This "process" view of an enumerator includes a wide range of constructs. Among the simplest are a counter for enumerating the positive integers in a given range and a pointer tracing down a linked list. Examples of more complex enumerators include processes for producing all possible propositional logic expressions or for producing the elements (of an unordered collection of integers) in increasing order. Note that some of these involve implicit (and even infinite) collections as well as explicit ones. PECOS's rules deal primarily with explicit collections.

Enumerating items in sequential collections (ENUMSEQ)

The principal **ENUMERATE-ITEMS** rule deals with enumerating the items in a sequential collection:

Rule ENUMSEQ.1:

> ***The items of a sequential collection may be enumerated by enumerating the locations in the collection and retrieving the items stored in each location[40].***

The internal representation of ENUMSEQ.1 is more complicated than the English form indicates, primarily because of a variety of optional property links, including "early exits" and an "initial action". These are omitted here in the interest of clarity.

As noted above, an enumerator can be seen as a coroutine that supplies items from the collection as they are needed. There are, of course, many ways to implement such a coroutine arrangement. PECOS's rules deal with only one of these, a generate and process structure. In such a structure, the actions to be performed between "calls" to the enumerator are embedded within the enumerator itself.

The following rule refines a location enumerator into a generate and process structure:

Rule ENUMSEQ.2:

> ***An enumeration of the locations in a sequential collection, with an action A to be performed for each location, may be implemented as a generate and process structure; the generator initialization is an initialization of the enumeration state; the generator incrementation is an incrementation of the enumeration state; the termination test is a test of whether the enumeration state is in its final state; the process consists of determining the next location from the enumeration state and performing the action A.***

----

[40] The retrieval of the items in the location is the **ELEMENT-AT-LOCATION** operation discussed in the section on sequential collections.

This rule includes two other parts that are most properly classified as temporary solutions ("kluges") to problems involving certain special cases:

Positions vs. locations: Recall that a distinction is made between locations (in which the elements are stored in a sequential collection) and positions (which occur "between" locations). ENUMSEQ.2, the enumeration rule for locations, is also used for enumerating positions (as needed for inserting elements into ordered sequential collections). A problem arises, however, since a collection with *n* locations has *n+1* positions, including those before the first location and after the last location. An enumeration of the locations would normally go through only *n* iterations before exiting. To allow for the *n+1*st position to be considered, the "process" is actually separated into two parts, and the determination of the next "location" from the enumeration state is done before the termination test. When the termination action is executed, the *n+1*st position is available if necessary. While this solution works in the situations in which it has been tested, it is clearly unsatisfactory: the relationship between enumerations of locations and positions would benefit from further analysis and clarification.

Special action for the first location: Many enumerations require that some special action be performed for the first location in the collection. For example, if the enumeration is part of a search for the location of the smallest element in the collection, the standard technique involves saving the location of the smallest element found so far. This must be initialized to the first location produced by the enumerator. If such a special action is specified, the process (in the GENERATE-AND-PROCESS) is initialized by determining the location from the initial state, performing the special action, and incrementing the state. While this solution seems adequate, it is not very satisfying. More flexible rules for coroutines would be helpful.

The next rule can now be used to refine the generate and process structure into a simple kind of loop:

Rule ENUMSEQ.3:

> ***A generate and process structure can be implemented as a loop with an exit test before the loop body; the loop initialization consists of the initial actions for the generator and the process; the body consists of the process followed by the generator incrementation; the exit test is the termination test of the generator.***

The rules for dealing with such loops will be covered in section 6.6. The different parts of an enumerator (initialization, incrementation, and termination test) are closely related, although they may be physically separated in the actual code. There are two fundamental decisions involved in constructing the parts: determining the order in which the items are to be produced and selecting a scheme for saving the state between calls to the enumerator. In the constructed code for the enumerator, there is nothing explicitly corresponding to these decisions. Rather they are implicit in the way the enumerator parts are coordinated so that they function properly together. The identification of these two decisions is a good example of the explication of "hidden" decisions involved in the programming process.

6.2.2. Enumeration order

As noted above, the enumeration order may be specified to be based on some ordering relation. If no enumeration order is specified, one must nonetheless be selected: an enumerator cannot be implemented without knowing the order in which the elements are to be enumerated. There are several possible orders in addition to those based on ordering relations on the elements. One very useful order is based on the fact that most structures have some "natural" order. With sequential collections, this natural order is the first-to-last order; either from the first cell to the last cell (for linked lists) or in order of increasing index (for array subregions). The order will be referred to as the "stored order" of a sequential collection.

Enumeration order (ENUMORDER)

In the absence of any reason to select otherwise, the stored order is usually a reasonable one to choose, and PECOS does this by applying the following rule:

Rule ENUMORDER.1:

> ***The locations in a sequential collection may be enumerated in the order in which they are stored.***

There is actually another way to have an enumeration order that is the same as the stored order:

Rule ENUMORDER.2:

> ***If the enumeration order is based on an ordering relation and the elements of a sequential collection are stored according to the same relation, the enumeration order is the same as the stored order.***

This rule is especially useful when EXPCOL.6, the membership rule involving an ordering relation, is used. In fact, a useful heuristic is that EXPCOL.6 should only be applied if the collection is kept ordered[41].

The stored order has an important property that may be described as "linearity": the enumeration order bears a simple relationship to the structure and is independent of the items that are actually stored in the locations. The following rule is used to answer queries about the linearity of the enumeration order:

Rule ENUMORDER.3:

> ***If the enumeration order is the same as the stored order, it is linear.***

The only other linear order is the reverse of the stored order (e.g., by decreasing index in an array subregion):

----

[41] Otherwise a membership test with order n^2 running time would be implemented!

Rule ENUMORDER.4:

If the enumeration order is the reverse of the stored order, it is linear.

Typically, nonlinear enumeration orders are based on ordering relations defined for the elements of the collection. For almost any type of element, an ordering relation could be defined. However, PECOS's rules currently deal with only one particular ordering relation:

Rule ENUMORDER.5:

An ordering relation for integers is "greater than".

For any ordering relation to be useful, it must be possible to compare two objects to see if one follows the other under the relation:

Rule ENUMORDER.6:

A test of whether an item X follows an item Y under the relation "greater than" may be implemented as a test of whether X is greater than Y.

In a few situations, PECOS also needs to be able to deal with the opposite relation, "less than". In particular, comparisons must occasionally be made:

Rule ENUMORDER.7:

A test of whether an item X follows an item Y under the relation "less than" may be implemented as a test of whether Y is greater than X.

PECOS'S rules do not make any special provisions for the case in which the two items may be the same (e.g., if a collection has repeated elements).

6.2.3. Enumeration state

One of the principal features of enumerators is that they produce each element of the collection exactly once[42]. This implies that there must be some way for the enumerator to "remember" which elements have been produced and which have not. That is, the state of the computation must somehow be saved. There are a variety of ways that such states can be saved. In the coroutine model, the state is saved within the control structure. In the generate and process model, there must be some data structure that represents the "state".

There are four operations that can be performed on such data structures:

----

[42] Unless, of course, the collection itself has repeated elements.

INITIAL-ENUMERATION-STATE
Returns a data structure representing the initial state of the enumeration; intuitively, the initial state means that "no locations have been produced."

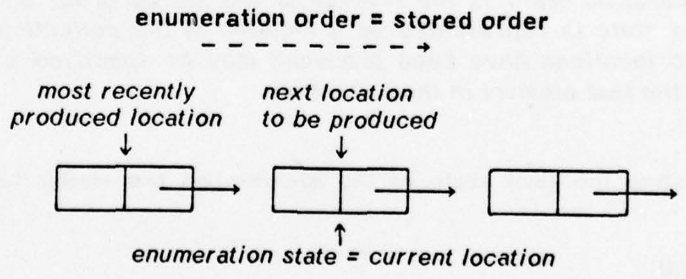
TEST-FINAL-ENUMERATION-STATE
Returns "True" if the enumeration state is in its final state and "False" if it is not; intuitively, the final state means that "all locations have been produced."

INCREMENT-ENUMERATION-STATE
Modifies the enumeration state to reflect the fact that a particular location (one of the arguments to this operation) has been produced.

FIND-ENUMERATED-LOCATION
Given an enumeration state, returns the next location to be enumerated; the implementation of such an operation will depend on the enumeration order.

6.2.3.1. Linear enumeration states

Whenever the enumeration order is "linear" (as described above), it is fairly easy to save the state of the enumeration. All that is needed is to remember the current location in the sequential collection (either the current cell or the current index). Although other possibilities exist, PECOS's rules assume that the current location is the next location to be produced. Thus, all of the locations before the current location (or after it, in the case of the reverse stored order) have already been produced, and the current location and all of the locations after it (or before, in the reverse case) have not yet been produced. The diagram below illustrates this for the case of a "stored order" enumeration of a linked list:


enumeration order = stored order
most recently produced location
next location to be produced
enumeration state = current location

Linear enumeration states (ENUMLINEAR)

The first rule is used to refine an enumeration state data structure into a location indicator:

Rule ENUMLINEAR.1:

> ***If the enumeration order is linear with respect to the stored order, the state of an enumeration may be represented as a location in the sequential collection.***

The first rule for initializing such states (the INITIAL-ENUMERATION-STATE operation) depends on the enumeration being "total" (i.e., all locations in the collection are to be produced):

Rule ENUMLINEAR.2:

> ***If an enumeration is total, the initial enumeration state must specify that no elements have been produced.***

Partial enumerations will be discussed later.

The next two rules deal with representing the fact that no elements have been produced in the two linear cases:

Rule ENUMLINEAR.3:

> ***If the enumeration order is the same as the stored order and the enumeration state is represented as a location in the collection, the fact that no locations have been produced may be specified by the location of the first element in the collection.***

Rule ENUMLINEAR.4:

> ***If the enumeration order is the reverse of the stored order and the enumeration state is represented as a location in the collection, the fact that no locations have been produced may be specified by the location of the last element in the collection.***

The rules for testing the final state of the enumeration are similar to those for initializing it:

Rule ENUMLINEAR.5:

> ***If an enumeration is total, a test of whether the enumeration state is in the final state may be implemented as a test of whether the state specifies that all of the locations have been produced.***

Rule ENUMLINEAR.6:

If the enumeration order is the same as the stored order and the enumeration state is represented as a location in the collection, a test of whether all of the locations have been produced may be implemented as a test of whether the location is the location after the last element of the collection.

Rule ENUMLINEAR.7:

If the enumeration order is the reverse of the stored order and the enumeration state is represented as a location in the collection, a test of whether all of the locations have been produced may be implemented as a test of whether the location is the location before the first element of the collection.

The rules for incrementing linear enumeration states also depend on whether the enumeration order is the stored order or the reverse stored order:

Rule ENUMLINEAR.8:

If the enumeration state is represented as a location in the collection, the state may be incremented by using the location of the next element to be produced.

Rule ENUMLINEAR.9:

If the enumeration order is the same as the stored order and the enumeration is total, the location of the next element to be produced is the location after the current location.

Rule ENUMLINEAR.10:

If the enumeration order is the reverse of the stored order and the enumeration is total, the location of the next element to be produced is the location before the current location.

Finally, the location must be produced from the current state. As shown in the diagram above, the current location is exactly the location to be produced, and the following rule reflects that:

Rule ENUMLINEAR.11:

If the enumeration state is represented as a location in the collection, the location to be produced may be implemented by simply returning the state indicator itself.

(Note that this rule is independent of whether the enumeration order is the stored order or its reverse.)

Partial enumerations (ENUMPART)

The rules given above deal with total enumerations: all of the elements (or locations) of the sequential collection are to be enumerated. For many purposes, partial enumerations are required. In the array shifting rules, for example, only the indices within a certain range are to be enumerated. The following rules deal with enumeration states for partial enumerations over array subregions[43]:

Rule ENUMPART.1:
If the enumeration order is the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, the initial enumeration state is the lower bound.

Rule ENUMPART.2:
If the enumeration order is the reverse of the the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, the initial enumeration state is the upper bound.

Rule ENUMPART.3:
If the enumeration order is the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, a test of whether the state is in its final state may be implemented as a test of whether the state is greater than the upper bound.

Rule ENUMPART.4:
If the enumeration order is the reverse of the the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, a test of whether the state is in its final state may be implemented as a test of whether the state is less than the lower bound.

When incrementing the enumeration state, the rule given above for linear enumeration states (rule ENUMLINEAR.8) would be applied, followed by one of the next two rules:

Rule ENUMPART.5:
If the enumeration order is the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, the location of the next enumerated item is the location after the current location.

----

[43] PECOS has no rules dealing with partial enumerations over linked lists.

Rule ENUMPART.6:

If the enumeration order is the reverse of the the stored order, the enumeration state is represented as an array index, and the range is determined by lower and upper bounds, the location of the next enumerated item is the location before the current location.

Note that the rule for retrieving the location from the current state (rule ENUMLINEAR.11 given earlier) is applicable for partial enumerations as well as for total enumerations. No special case rule is needed.

6.2.3.2. Operations applied to locations in collections

The linear enumeration state rules have introduced several location operations other than the ELEMENT-AT-LOCATION operation discussed earlier. The operations needed for "stored order" enumerations are as follows:

LOCATION-OF-FIRST-ELEMENT
: Returns the location of the first element in the collection.

TEST-LOCATION-AFTER-LAST-ELEMENT
: Tests whether a location is the location after the last element in the collection.

LOCATION-AFTER-LOCATION
: Returns the location after a given location.

The following are needed for "reverse stored order" enumerations:

LOCATION-OF-LAST-ELEMENT
: Returns the location of the last element in the collection.

TEST-LOCATION-BEFORE-FIRST-ELEMENT
: Tests whether a location is the location before the last element in the collection.

LOCATION-BEFORE-LOCATION
: Returns the location before a given location.

Location operations (LOCOP)

For locations in linked lists, only the "stored order" operations are relevant. There is no simple way to enumerate the cells of a singly linked list in the reverse order. Since there are two ways to represent linked lists (with or without special header cells) and two ways to represent locations in linked lists (a link to the cell of the location or a link to the cell preceding the location), the rules for dealing with these location operations must deal with a variety of cases. In the interest of clarity, slightly simplified versions of the rules are presented here. There are three rules dealing with the location of the first item:

Rule LOCOP.1:

If a linked list is represented as linked free cells without a header cell, and if a location is represented as a link to the cell of the location, the location of the first item may be returned by returning a link to the first cell in the list.

Rule LOCOP.2:

If a linked list is represented as linked free cells with a header cell, and if a location is represented as a link to the cell of the location, the location of the first item may be returned by returning the link part of the first cell in the list.

Rule LOCOP.3:

If a linked list is represented as linked free cells with a header cell, and if a location is represented as a link to the cell preceding the cell of the location, the location of the first item may be returned by returning a link to the first cell in the list.

Note that there is no rule for the case of a location represented by a predecessor link when the list does not have a special header cell. The problem, of course, is that there is no cell before the cell of the first element.

Testing whether the location is the location after the last item differs only with respect to the location representation, but not with respect to the existence of a header cell:

Rule LOCOP.4:

If a location is represented as a link to the cell of the location, a test of whether the location indicates the location after the last item may be implemented by a test of whether the location indicates the empty list flag[44].

Rule LOCOP.5:

If a location is represented as a link to the cell preceding the cell of the location, a test of whether the location indicates the location after the last item may be implemented by a test of whether the link part of the cell indicated by the location is the empty list flag.

----

44 The TEST-EMPTY-FREE-LINK operation discussed earlier.

The process of determining the location after a given location is the same for both location representations:

Rule LOCOP.6:

If a location is represented as a link to the cell of the location, the location after a given location may be returned by retrieving the link part of the cell indicated by the location[45].

Rule LOCOP.7:

If a location is represented as a link to the cell preceding the cell of the location, the location after a given location may be returned by retrieving the link part of the cell indicated by the location.

For array subregions, is is quite feasible to enumerate the locations in either the stored order (increasing index) or the reverse stored order (decreasing index), and PECOS's rules deal with both cases:

Rule LOCOP.8:

If a location is represented as an index in an array subregion, the location of the first item may be returned by returning the index of the lower bound of the subregion.

Rule LOCOP.9:

If a location is represented as an index in an array subregion, the location of the last item may be returned by returning the index of the upper bound of the subregion.

Rule LOCOP.10:

If a location is represented as an index in an array subregion, a test of whether the location indicates the location after the last item in the subregion may be implemented as a test of whether the location index is greater than the index of the upper bound.

Rule LOCOP.11:

If a location is represented as an index in an array subregion, a test of whether the location indicates the location before the first item in the subregion may be implemented as a test of whether the index of the lower bound is greater than the location index.

----

[45] The LINK-OF-CELL operation discussed earlier.

Rule LOCOP.12:

> ***If a location is represented as an index in an array subregion, the location after a given location may be returned by returning the sum of the location index and the integer 1.***

Rule LOCOP.13:

> ***If a location is represented as an index in an array subregion, the location before a given location may be returned by returning the difference between the location index and the integer 1.***

6.2.3.3. Nonlinear enumeration states

Unless the elements are stored in order, an enumeration order based on an ordering relation is nonlinear: the temporal order in which the elements are produced does not bear any simple relationship to the physical order in which they are stored. Nonlinear orderings generally lead to more complex enumerators, since both finding the next element (location) to be produced and saving the state are not as simple as with linear orders. Finding the next element to produce typically involves searching through the entire collection to find the element that comes next according to the ordering relation. In the process, the state must be interrogated to determine whether an element has already been produced, so that it will not be produced again. There are basically two types of techniques that can be used to save the enumeration state for nonlinear enumerations: destructive and nondestructive. With destructive schemes, each element is somehow removed from the collection after it is enumerated. The two obvious techniques for performing the removal are to remove the location in which the element is stored and to overwrite the element with some special marker. In the first case, all locations in the collection contain elements that have not yet been produced, so the "interrogation" part of the search is actually unnecessary. In the second case, the interrogation of the state is performed by testing whether the item in the location is the special marker. There are several nondestructive schemes. Abstractly, all that is necessary is to maintain some kind of mapping between elements (or locations) and indicators of whether or not the element (or location) has already been produced. The search process can then interrogate the state by retrieving the image of the element under the mapping. Under this view, all of the knowledge relevant to mappings could be applied here. For example, a hash table of locations could be maintained, or the mapping could be inverted so that, in effect, the set of enumerated elements would be stored. Unfortunately time restrictions prevented this topic from being explored to any great detail, and PECOS's rules do not cover any nondestructive enumeration state techniques. In fact, the only nonlinear technique covered by PECOS's rules is the deletion of the enumerated locations from the collection.

Enumeration state-saving through deletion (ENUMDEL)

The first rule for deletion involves a test of whether the collection can be destroyed:

Rule ENUMDEL.1:

> *If a collection is destructible, the enumeration state may be saved by deleting locations from the collection.*

The question, of course, is how to determine whether a collection can be destroyed. PECOS's rules are relatively conservative in that respect. There is only one rule for determining destructibility:

Rule ENUMDEL.2:

> *A collection is destructible if it has at some point been explicitly noted as being destructible.*

And there is only one rule that makes such an explicit note:

Rule ENUMDEL.3:

> *The items of a collection may be enumerated by duplicating the collection and enumerating the items of the new collection; the new collection is destructible*[46].

ENUMDEL.3 is another rule for refining ENUMERATE-ITEMS constructs. Note that ENUMDEL.3 is applicable whether or not the enumeration order is nonlinear. It would be correct to use a destructive scheme even in the linear case. One of PECOS's choice-making heuristics, however, suggests choosing ENUMDEL.3 only if the enumeration order is known to be nonlinear.

The rules for manipulating the enumeration state are relatively simple.

Rule ENUMDEL.4:

> *If the enumeration state is saved by deleting locations from the collection, the fact that no locations have been produced may be specified by using the original collection.*

Rule ENUMDEL.5:

> *If the enumeration state is saved by deleting locations from the collection, a test of whether the state specifies that all of the locations have been produced may be implemented as a test of whether the collection is empty.*

----

[46] Techniques for duplicating collections were discussed earlier (section 6.1.5).

Rule ENUMDEL.6:

> ***If the enumeration state is saved by deleting locations from the collection, the state may be incremented by removing the item at the location that has most recently been produced.***

The most complicated aspect of nonlinear enumerations through deletion is the operation of finding the next location to produce. As noted above, the next location to produce is that of the least element according to the ordering relation:

Rule ENUMDEL.7:

> ***If the enumeration state is saved by deleting locations from the collection, and the enumeration order is ordered by a relation R, the next location to be produced may be determined by finding the location of the least element according to R.***

The process of finding this location is normally done by considering all of the locations to determine the location which contains least element. The standard technique involves saving a pointer to the "least so far", which must be initialized to the first location that is considered. After this initialization, the elements at the rest of the locations are compared with the "least so far", replacing it if the new element is found to be smaller. The reasoning process that leads to this technique is fairly complex **[Green and Barstow 1977b]**. In PECOS's rule base, all of this complexity is encoded in a single rule. A more detailed set of rules for this situation would be a valuable extension.

Rule ENUMDEL.8:

> ***The location of the least element (according to a relation R) may be found by enumerating the locations of the collection; for the first location, remember the location as BEST; for the other locations, the action to be performed is a test of whether the element at BEST follows the element at the location according to R; if it does, remember the new location as BEST; after all locations have been considered, return BEST as the result.***

Note that ENUMDEL.8 specifies a special action to be performed for the first location. The relationship of this case to the basic enumeration structure was discussed earlier (as one of the "kluges" in the enumeration rule). From this point on, the rules for enumerating locations can be applied, and the entire nonlinear enumeration is implemented as an enumeration within an enumeration.

6.2.4. Other enumeration operations

There are several other operations involving enumerations over stored collections. As they do not fit conveniently into any of the categories previously discussed, they are included here.

<u>Other enumeration operations (ENUMMISC)</u>

Recall that the process of adding an element to an ordered sequential collection involves finding the particular position (i.e., the space between two locations) in which the new element belongs. There are several ways that this may be done. For array subregions, for example, some variant of binary search might be faster than a linear scan[47]. PECOS's rules, however, deal only with a simple linear enumeration:

<u>Rule ENUMMISC.1</u>:

> ***The operation of finding the appropriate position for an element in an ordered sequential collection may be implemented by enumerating the the locations of the collection; the action performed for each location is to test whether the item at the location follows the new element according to the ordering relation; if so, halt the enumeration and return that location; if the enumeration runs to completion, then the appropriate position is the last position in the collection.***

(This rule uses the "kluge" for relating position and location enumerations, as discussed earlier. A cleaner set of rules for dealing with this situation would be a worthwhile extension.)

The operation of removing an element from a sequential collection required finding the location of the element. The following rule may be used to implement the search for that location:

<u>Rule ENUMMISC.2</u>:

> ***The operation of finding the location of an item in a sequential collection may be implemented by enumerating the locations of the collection; for each location, if the item at the location is the desired item, halt the enumeration and return the location found.***

(Since the removal operation assumes that the element is in the collection originally, there is no need to deal with the case in which no location satisfies the test.)

Under some circumstances, the position of an element in a collection can be deduced while the program is being constructed. One such case is that in which the element was originally determined by retrieving the item at some location:

----

[47] But since the insertion of the new element still requires a scan for the shift, the find plus insertion still runs in linear time.

Rule ENUMMISC.3:

> ***If an element X was determined by retrieving the item at a location L of a sequential collection C the location of X in C is L.***

If the location can be deduced, there is no need for a search:

Rule ENUMMISC.4:

> ***If the location of an item X in a sequential collection C is known to be L, an operation of finding the location of X in C can be implemented by simply using L.***

These two rules illustrate a style different from most of the other rules: they deal with the use of state information to simplify or avoid a computation. These rules were not particularly easy to express in PECOS's rule formalism. In fact, five separate rules were needed in order to trace certain kinds of pointers to try to deduce the necessary information. The development of better techniques for using state information could be a valuable extension. Perhaps what is needed is a more general mechanism that tries to deduce the result of any part of the program before writing code to compute the result.

The final kind of enumeration that PECOS can construct involves enumerating the elements in a collection represented as a Boolean mapping:

Rule ENUMMISC.5:

> ***The items in a collection represented as a Boolean mapping may be enumerated by enumerating the inverse image of "True" under the mapping.***

The next rule applies to any attempt to enumerate the items in an inverse image of association table mappings, and is not restricted to Boolean mappings:

Rule ENUMMISC.6:

> ***If a mapping is represented as an association table, the inverse image of a range element R may be enumerated by enumerating the keys of the table and considering only those keys whose associated value is R.***

Rule ENUMMISC.7:

> ***The keys of an association tables represented as an array may be enumerated by enumerating the integers between the lower and upper bounds.***

The final rule in this chain is also applicable whether or not the range elements are Boolean values:

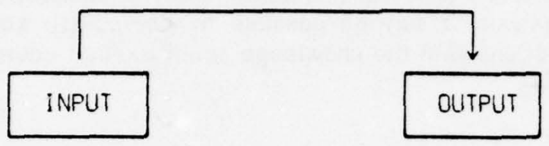
Rule ENUMMISC.8:

An enumeration of the integers between a minimum value X and a maximum value Y in the order of increasing value may be implemented as a generate and process structure; the generator initialization sets the state to X; the generator incrementation increments the state by 1; the termination test is a test of whether the state is greater than Y.

Note that this rule simply constructs a generate and process for an implicit collection, rather than for an explicit collection as we have seen earlier. The "enumeration order" is in terms of increasing value, and the enumeration state is an integer. With further analysis, it may be possible to incorporate knowledge about enumerating implicit collections with the knowledge about explicit collections.

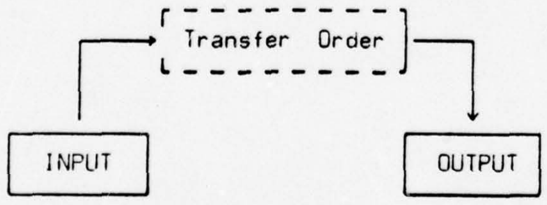
6.3. The transfer paradigm for sorting

The rules given up to this point provide much of the programming knowledge needed for some simple sorting algorithms. One class of sorting algorithms may be described as transfer sorts. Algorithms in this class take a collection as input and produce another collection as the output. Both collections are represented as sequential collections, and the elements of the output are required to be stored according to some ordering relation. At each stage in a transfer algorithm, one element is selected from the input and added to the output[48]. The process is illustrated below:



Transfer Paradigm

Algorithms in this class divide naturally into two categories, selection sorts and insertion sorts. In selection sorts, the elements are selected from the input in the same order as they are to be stored in the final output (i.e., according to the same ordering relation). The part of the program that performs the selection is relatively complicated. Since the elements arrive in the same order in which they are to be stored, the part that adds them to the output set is relatively simple, usually an addition at either the front or the back. With insertion sorts, the elements are selected from the input in any convenient order (normally the stored order) and added to the output in such a way that the output at each stage is in the correct order. Thus, the part that adds the element to the output is relatively complicated, involving a search for the correct position to add the element. The part that does the selecting, on the other hand, is relatively simple. The essential difference between these two categories involves the order in which the elements are transferred from the input to the output. This order will be referred to as the *transfer order* and is illustrated below:



In a selection sort, the transfer order bears a simple linear relationship to the desired order for the output collection (the sorted order). In an insertion sort, the transfer order is linearly related to the stored order of the input.

----

48 Note that the running time of these algorithms is $O(n^2)$.

With a few modifications, the transfer sorting algorithms discussed here can be adapted to perform certain kinds of "in-place" sorts, algorithms in which the input and output collections are stored in the same structure. For example, an array to be sorted may divided (conceptually) into two regions, an "input" which is the unsorted part and an "output" which is the sorted part. Under this view, an insertion sort becomes the classical bubble sort and a selection sort becomes a sinking sort [Knuth 1973]. The reader is referred elsewhere for a more detailed discussion of these issues, as well as some aspects of more sophisticated sorting programs such as quicksort and mergesort [Green and Barstow 1977b].

Transferring elements in sequential collections (TRANSFER)

The importance of the transfer order can be seen in the first two rules, each providing one way to choose the transfer order[49]:

Rule TRANSFER.1:

> *A transfer order for a transfer operation is the stored order of the input collection.*

Rule TRANSFER.2:

> *A transfer order for a transfer operation is the stored order of the output collection.*

The first of these two rules leads to an insertion sort and the second leads to a selection sort. Once the transfer order has been chosen, the actual transfer operation can be refined:

Rule TRANSFER.3:

> *A transfer of the elements from one sequential collection to another may be implemented by a total enumeration of the items in the input collection, where the enumeration order is the transfer order; the action for each item is to add it to the output.*

When the transfer order is the stored order of the input, then the addition operation will require searching for the right position and adding it there. The rules for adding elements to ordered collections were given earlier. When the transfer order is the same as the sorted order, however, this information is not used very effectively, since the addition operation is considered essentially in isolation. The following rule can be used to take fuller advantage of the transfer order:

----

[49] Note that these rules, as well the other rules in this section, are expressed in terms of sequential collections: they are equally applicable for linked lists and for array subregions.

Rule TRANSFER.4:

> ***If the transfer order is the same as the stored order of the output, a transfer of the elements from one sequential collection to another may be implemented by a total enumeration of the items in the input collection, where the enumeration order is the transfer order; the action for each item is to add it to the back of the output[50].***

The use of two separate rules for refining the transfer operation is aesthetically unsatisfying, but they are probably both necessary until better methods for representing temporal constraints (as well as coroutine structures) have been incorporated into PECOS's program description formalism. In this particular case, the obvious way to merge the two rules is to describe the action (addition of the element to the output) as a process similar to an enumerator, and to specify that the elements will arrive in the transfer order. Various methods for describing such constructs have been considered, but PECOS currently does not use any of them [Green and Barstow 1975].

----

[50] It is interesting to note that this rule also correctly handles the case in which the input collection is already sorted. The enumeration rules determine the "linearity" of an enumeration order by comparing it to the order in which the elements are stored. If the input is already sorted, the enumeration order is the same as the stored order, so a simple scan of the input will be constructed.

6.4. Mappings

A mapping is a way of associating objects in one set (range elements) with objects in another set (domain elements). In PECOS's rules, the domain and range sets are only implicit, with generic descriptors of domain elements and range elements being stored as part of the mapping descriptor. In addition, PECOS only deals with many-to-one mappings and not with more general correspondences or relations. The range element to which a given domain element maps will be referred to as the image of the domain element. The set of domain elements that map to a given range element will be referred to as the inverse image of the range element. A mapping may (optionally) have a default image: if there is no stored image for a particular domain element, a request to determine its image can return the default image. For example, when a Boolean mapping is used to represent a collection, the default image is "False."

There are seven basic operations for dealing with mappings:

NEW-MAPPING
Creates a new mapping and returns it as the operation's value. A list of <domain element, range element> pairs to be contained in the mapping initially may also be specified.

STORE-IMAGE
Sets the image of a given domain element under a given mapping to a given range element.

CHANGE-IMAGE
Changes the image of a given domain element under a given mapping from one given range element to another given range element.

IS-IMAGE
Tests whether a given range element is the image of a given domain element under a given mapping.

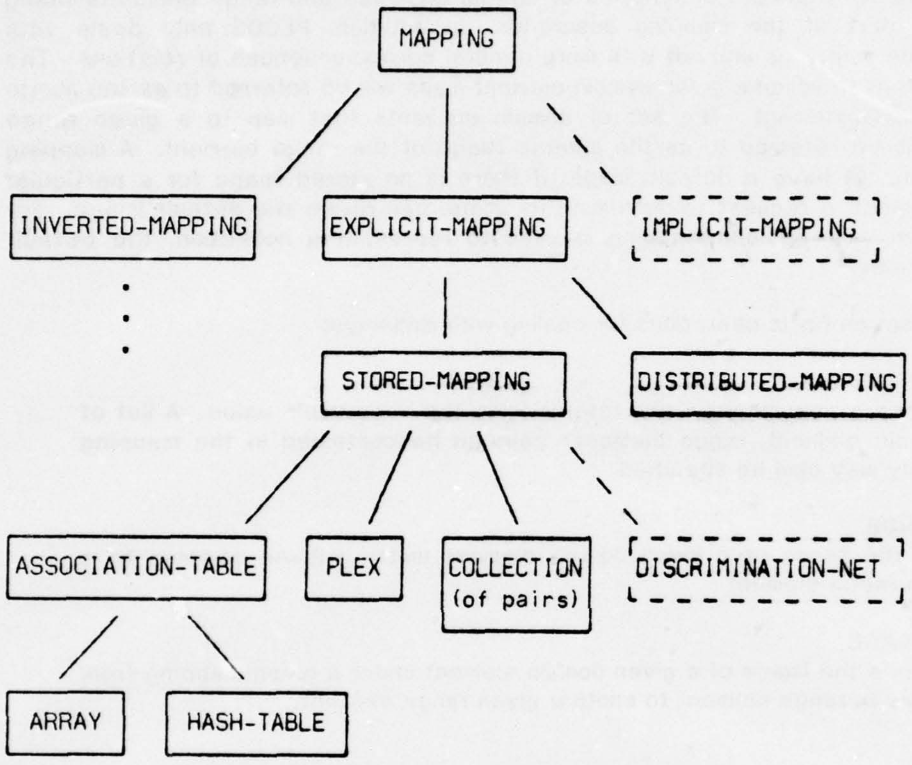
GET-IMAGE
Retrieves the range element that is the image of a given domain element under a given mapping.

GET-INVERSE-IMAGE
Returns a collection whose elements are all the domain elements whose image is a given range element under a given mapping.

DOMAIN-OF-MAPPING
Returns a collection whose elements are all the domain elements for which any image exists.

6.4.1. Overview of mapping representations

The following diagram summarizes representation techniques for mappings.



6.4.2. Rules about mappings

Explicit Mappings (EXPMAP)

Just as with collections, a mapping may be represented either explicitly or implicitly. An explicit mapping is one in which every pair is somehow given explicitly. In an implicit mapping the pairs are computed as needed. For instance, a function that computes X+1 given X is an implicit mapping. In addition to the pure cases, there is also the possibility of a hybrid representation. For example, since "False" is the default image for a Boolean mapping, it is an implicit representation for any domain element that does not have an explicitly stored image. An interesting variant of the Reachability Program involves representing the MARKS mapping with two explicit range elements ("EXPANDED" and "BOUNDARY") and one implicit range element ("UNEXPLORED"). If such a representation is used, the initialization of the mapping with all vertices mapped to "UNEXPLORED" is unnecessary, and the test of whether a vertex is "UNEXPLORED" must be implemented as a test of whether the image is neither "BOUNDARY" nor "EXPANDED".

PECOS's rules currently deal only with explicit mappings:

Rule EXPMAP.1:

> ***A mapping may be represented explicitly.***

As with collections, there are two rules dealing with the creation of instances of a mapping. Typically, a new instance of a mapping is created without specifying any initial pairs, and the following rule deals with this case:

Rule EXPMAP.2:

> ***If a mapping is represented explicitly, a new mapping with no initial pairs may be created by creating a new explicit mapping with no initial pairs.***

The second rule for creating new instances of mappings deals with the case in which the list of initial pairs is non-empty:

Rule EXPMAP.3:

> ***A mapping with a non-empty list of initial pairs may be created by first creating a mapping with no initial pairs and then, for each pair in the list, storing the range part as the image of the domain part.***

As with collections, one must careful to distinguish between knowing the list of initial pairs at compile time and at run time. PECOS's specification language requires an explicit list of pairs of operations, and EXPMAP.3 applies in this case. To specify a variable list of initial pairs, one would have to specify something like a collection of plexes (with domain and range parts) and use a **FOR-ALL-DO** construct to explicitly store the images for each plex of the collection.

Most of the other operation refinement rules for explicit mappings simply parallel the data structure refinement and are omitted. The one exception is the rule for testing whether a given range element is the image of a given domain element (an **IS-IMAGE** test):

<u>Rule EXPMAP.4</u>:

A test of whether a given range element is the image of a given domain element may be computed by retrieving the image of the domain element and testing whether it is equal to the given range element.

The principal value of this rule is that an **IS-IMAGE** test has been refined into a GET-IMAGE operation, so that the **IS-IMAGE** test need not be refined to all levels for all representations of mappings.

There are two ways to represent mappings explicitly: storing the associations in a single structure or storing them in some kind of distributed structure. The rules for distributed structures will be presented later. The following rule is for stored mappings:

<u>Rule EXPMAP.5</u>:

An explicit mapping may be stored in a single structure.

The operation refinement rules parallel the data structure refinement rule.

There are many structures in which associations can be stored. Most of these are dependent on the nature of the domain of the mapping. One interesting case (unfortunately, not covered by the current rule set) is the use of a discrimination net. The codification of knowledge about discrimination nets would be a very useful extension of the work on PECOS. PECOS can successfully deal with several other structures.

<u>Mappings as collections of pairs (PAIRS)</u>

One way to store the associations of an explicit mapping is with a collection of <domain element, range element> pairs. Each element of the collection specifies a particular association. The following rule introduces this representation:

<u>Rule PAIRS.1</u>:

A stored mapping may be represented as a collection whose elements are pairs with a domain part and a range part.

Since the mapping has been refined into a collection, all of the collection rules may be applied here. If the collection is represented as a linked list, the familiar association list structure results. Note also that there are no restrictions on the domain of the mapping.

The other rules for this representation refine the operations on mappings into operations on collections:

Rule PAIRS.2:

If a stored mapping is represented as a collection, a new instance of the mapping (with no initial pairs) may be created by creating a new collection with no initial elements.

Rule PAIRS.3:

If a stored mapping is represented as a collection, a given range element, R, may be stored as the image of a given domain element, D, by adding the pair <D, R> to the collection.

Rule PAIRS.4:

If a stored mapping is represented as a collection, the image of a given domain element, D, may be changed from a given range element, R_1, to a given range element, R_2, by finding in the collection an element whose domain part is D, and replacing its range part by R_2; if no such element is found, the pair <D, R_2> should be added to the collection.

Rule PAIRS.5:

If a stored mapping is represented as a collection, M, the inverse image of a given range element, R, may be computed by first creating a collection, C, with no initial elements and then, for all elements, X, in M, such that the range part of X is R, add the domain part of X to C.

Rule PAIRS.6:

If a stored mapping is represented as a collection, M, the domain of the mapping may be computed by first creating a collection, C, with no initial elements and then, for all X in M, add the domain part of X to C.

Rule PAIRS.7:

If a stored mapping is represented as a collection, the image of a given domain element, D, may be computed by finding in the collection an element whose domain part is D, and returning its range part; if no such element is found, return the default image.

Mappings with separate fields for each domain element (PLEXMAP)

Most of the representation techniques for mappings have dealt with domain elements that are instances of some general description. Sometimes the domain of a mapping is a fixed set of known alternatives. For example, in the inverted mapping $\text{MARKS}_{\text{inv}}$ in the Reachability Program, the domain collection is precisely the collection {"EXPANDED", "BOUNDARY", "UNEXPLORED"}. Under such circumstances, one can create some kind of record structure (in PECOS's terms, a plex) with one field for each domain element and store the image in that field[51]. The following rule applies in such situations:

Rule PLEXMAP.1:

A stored mapping whose domain is a fixed set of alternatives and whose typical range element is Y may be represented as a plex with one field for each alternative and with each field being Y[52].

An important observation to make is that one is no longer concerned with a generic descriptor of the range elements of the mapping: there is now a separate descriptor for each possible range element (each image of one of the known domain elements). With a single generic descriptor, all of the range elements must necessarily be represented in the same way. But with a separate descriptor for each range element, this restriction no longer applies. Recall, for example, that "UNEXPLORED" in the Reachability Program was represented as a Boolean array, while "BOUNDARY" and "EXPANDED" were represented as linked lists.

The rule for creating a new instance of such a mapping shows this feature:

Rule PLEXMAP.2:

If a mapping is represented as a plex with a field for each domain element, a new instance of the mapping may be created by creating a new instance of the plex, with each field being a new instance of the descriptor of that field.

While this ability to represent different range elements differently has certain efficiency advantages, it causes complications for some of the operation refinement rules. If the domain element is known at compile time, the rules are fairly simple:

----

[51] This is useful only if the domain collection is fairly small.

[52] The rules in this section differ slightly from those in PECOS's implementation in order to focus on the programming knowledge involved, without getting lost in certain idiosyncrasies of the implementation.

Rule PLEXMAP.3:

If a mapping is represented as a plex with a field for each domain element, the image of a particular domain element D (known at compile time), may be retrieved by retrieving the field for D.

Rule PLEXMAP.4:

If a mapping is represented as a plex with a field for each domain element, a given range element, R, may be stored as the image of a particular domain element D (known at compile time), by storing R in the field for D.

Rule PLEXMAP.5:

If a mapping is represented as a plex with a field for each domain element, the image of a particular domain element D (known at compile time), may be changed from R_1 to R_2 by replacing the field for D by R_2.

Computing an inverse image under such a mapping is somewhat more complicated, and PECOS has no rules for dealing with it.

When the domain element is not known at compile time, most of the operations must be implemented as CASE statements, testing for each possible domain element, and dealing with the appropriate field in each case:

Rule PLEXMAP.6:

If a mapping is represented as a plex with a field for each domain element, the image of a given domain element D (unknown at compile time), may be retrieved by testing, for each D_i in the domain, whether D is equal to D_i, and if so, returning the field for D_i.

Note that the complete set of D_i is known at compile time, so that a case structure can be built.

Rule PLEXMAP.7:

If a mapping is represented as a plex with a field for each domain element, a given range element R may be stored as the image of a given domain element D (unknown at compile time), by testing, for each D_i in the domain, whether D is equal to D_i, and if so, storing R in the field for D_i.

Rule PLEXMAP.8:

If a mapping is represented as a plex with a field for each domain element, the image of a given domain element D (unknown at compile time), may be changed from R_1 to R_2 by testing, for each D_i in the domain, whether D is equal to D_i, and if so, replacing the field for D_i by R_2.

Association tables (ASSOCTABLE)

A common way to store associations in an explicit mapping is to use some kind of association table, where the keys in the table are the domain elements and the entries are the range elements. One of the principal features of such a table is that storage and retrieval time are roughly constant. On the other hand, it may be difficult to compute the domain or an inverse image.

The following is the data structure refinement rule for association tables:

Rule ASSOCTABLE.1:

> *A stored mapping with typical domain element X and typical range element Y may be represented with an association table whose typical key is X and whose typical value is Y.*

The operation refinement rules are all parallel, except that a **CHANGE-IMAGE** operation refines into a **STORE-IMAGE** operation, since the old entry in the table is simply overwritten:

Rule ASSOCTABLE.2:

> *If a stored mapping is represented as an association table, the image of a given domain element D may be changed from R_1 to R_2 by storing R_2 as the image of D.*

Rule ASSOCTABLE.3:

> *If stored mapping is represented as an association table, the inverse image of a range element R may be computed by a sequence of two operations: first, initialize a collection with no elements; then, enumerate the keys of the table, adding each to the collection if its image is R.*

Arrays as association tables (ARRAYTABLE)

One common way to represent an association table when the keys are integers is to use an array:

Rule ARRAYTABLE.1:

> *An association table whose typical key is an integer from a fixed range and whose typical value is Y may be represented as an array with typical entry Y.*

Note that the range of possible key values must be fixed, since they will provide the bounds on the array. In fact, the situation is even more complicated, since some languages only support arrays whose lower bound is 1. In such a case, if the lower bound of the fixed range is not 1, some kind of offset or conversion must be included.

The operation refinement rules simply parallel the data structure refinement rule. The refined operations involve the same operations mentioned earlier in connection with array subregions: NEW-ARRAY, DEPOSIT-IN-ARRAY, and RETRIEVE-FROM-ARRAY.

(Rules for enumerating the keys of an association table represented as an array are discussed briefly in connection with the enumeration rules in section 6.2.)

Hash tables (HASHTABLE)

When the keys of an association table are not integers (or the range is too large), a common representation technique is to use a hash table. PECOS's rules currently deal with hash tables only in a very limited sense: they are sufficient to utilize INTERLISP's hash array functions. PECOS's data structure refinement rule is as follows:

Rule HASHTABLE.1:

> ***An association table whose typical value is Y may be represented as a hash table with typical entry Y.***

Again, the operation refinement rules parallel the data structure rule.

In the more general notion of a hash table, there are actually two mappings involved. The first (a hashing function) maps keys of the association table into integer values in a given range, and the second (an array) maps those integers into entries. The desired association table is then implemented as the composition of the two mappings[53]. One complication with hash tables is that the first mapping is typically many-to-one, and two keys may map to the same integer although they are supposed to map to separate entries in the association table. Some technique must be used to prevent both from mapping to the same entry. Two common techniques used to perform this collision resolution are rehashing and the use of buckets (instead of single range objects) as the array entries. Much more knowledge about hash tables must be codified before they can be dealt with adequately by an automatic programming system.

----

[53] In fact, the notion of implementing a mapping as the composition of two mappings is more general than its use here for hash tables.

Distributed mappings (DISTMAP)

A distributed mapping is one in which the associations are not stored in a single structure. A simple example of a distributed mapping is to store the images on the property lists of the atoms in the domain of the mapping. Distributed mappings are introduced with the following rule:

Rule DISTMAP.1:
An explicit mapping may be distributed among the domain elements.

The operation refinement rules are parallel except that the domain and inverse image operations are not effectively computable.

Distributed mappings are the only kind of distributed data structure that PECOS can deal with[54]. One of the interesting questions is what it means to return a "distributed" structure as the value of an operation. Clearly, one cannot return the structure itself, since it doesn't exist as a single entity. The solution adopted in PECOS's rules is to pass the "name" of the structure. In the case of LISP property list markers, the name is the "property name". This example will be pursued further in the discussion on LISP rules in section 6.7.

Inverted Mappings (INVMAP)

Most representations for mappings have a "one way" flavor: it is generally easier to compute the image of a given domain element than to compute the inverse image of a given range element. For many applications, however, inverse images may be computed quite frequently. For example, in the Reachability Algorithm (see section 2), the inverse image of "BOUNDARY" under the MARKS mapping is a central part of the algorithm. In such cases, it is often useful to "invert" the mapping: rather than associating a range element with each domain element, a collection of domain elements may be associated with each range element. The following rule accomplishes this inversion:

Rule INVMAP.1:
A mapping with typical domain element X and typical range element Y may be represented as a mapping with typical domain element Y and typical range element a collection with typical element X; the default image under the inverted mapping is the empty collection[55].

----

[54] Distributed collections can be handled as distributed Boolean mappings (e.g., with "True" or "False" as the value of some property name for atoms that might be in the collection).

[55] PECOS's rule actually has an additional condition specifying that the range objects of the original mapping must be primitive (integer or string). This more properly belongs in a choice-making heuristic that reflects the fact that PECOS's current rule set can only implement mappings whose domain objects are primitive.

Note that the result of applying this rule is simply another mapping. This means that all of the techniques applicable to general mappings can be applied to the inverted mapping.

Since the inverse images are kept explicitly, many of the mapping operations manipulate collections. Storing an image is implemented by adding the domain element to the inverse image of the range element:

Rule INVMAP.2:

> *If a mapping is inverted, Y may be stored as the image of X by adding X to the image of Y under the inverted mapping.*

Changing an image requires removing the domain element from one collection and adding it to another:

Rule INVMAP.3:

> *If a mapping is inverted, the image of X may be changed from Y to Z by removing X from the image of Y under the inverted mapping and adding X to the image of Z under the inverted mapping.*

Testing whether a given range element is the image of a given domain element is implemented as a membership test:

Rule INVMAP.4:

> *If a mapping is inverted, a test of whether Y is the image of X may be implemented as test of whether X is a member of the image of Y under the inverted mapping.*

Computing the inverse image of a range element under the original mapping is quite simple:

Rule INVMAP.5:

> *If a mapping is inverted, the inverse image of Y may be computed by retrieving the image of Y under the inverted mapping.*

Certain operations are much more difficult using an inverted mapping (and PECOS has no rules to deal with them). In particular, to compute the image of a domain element requires enumerating all domain objects of the inverted mapping to test for membership, and computing the domain of the original mapping requires computing the union of all range elements under the inverted mapping. In situations where both images and inverse images are computed frequently, it may be advisable to consider multiple representations for the mapping.

6.4.3. Other mapping operations

<u>Other mapping operations (MAPMISC)</u>

Just as with collections, it is also possible to duplicate mappings. Again, the term "duplicated" refers to the fact that at the abstract level both objects have the same type, "mapping". However, at more refined levels, the two mappings may have different representations. PECOS can only duplicate one kind of mapping:

<u>Rule MAPMISC.1</u>:

> ***If a mapping is represented as a stored collection of pairs, it may be duplicated by a sequence of two actions: first, initialize the new mapping with no associations; then, for each element of the collection, set the image (under the new mapping) of the domain field of the pair to be the range field.***

6.5. Input, output, and representation conversion

The most interesting part of PECOS's abilities at writing input and output routines are the facilities for converting from one representation into another[56]. Every specification of an input operation includes a descriptor of the kind of representation that will be input. Similarly, every output operation includes a descriptor of the representation that should be produced. In particular, collections may be input or output either as linked lists without header cells or as array subregions. Mappings may be input or output only as lists of <domain, range> pairs (i.e., association lists). These representations may not be well suited to the particular way that the data structures are used. In the SUCCESSORS mapping of the Reachability Program, for example, the use of an association list requires that some searching be done within the inner loop of the algorithm. In such cases, it may be useful to convert the input representation into a different internal representation (or the internal into the output representation)[57].

Representation conversion (CONVERT)

PECOS has two rules for introducing representation conversions into input operations:

Rule CONVERT.1:

If a collection is input, its representation may be converted into any other representation before further processing.

Rule CONVERT.2:

If a mapping is input, its representation may be converted into any other representation before further processing.

Similarly, there are two rules for introducing representation conversions into output operations:

Rule CONVERT.3:

If a collection is output, its representation may be converted into any other representation before outputting it.

----

[56] These issues also arise whenever a data structure representation is constrained externally, not just when performing input and output.

[57] Of course, the cost of performing the conversion must also also be taken into account when assessing the utility of using a different internal representation.

Rule CONVERT.4:

If a mapping is output, its representation may be converted into any other representation before outputting it.

These rules actually seem overly restricted. In the long run, what is probably needed is some technique for considering the entire history of a data structure, introducing representation conversions wherever appropriate. In fact, for some purposes, multiple representations may be useful (e.g., representing a collection with both a linked list and property list markings), maintaining their consistency throughout the program that uses them.

Once the possibility of a conversion has been introduced, the techniques for performing the conversion are relatively straight-forward. One rule is needed for the case in which the two representations are the same:

Rule CONVERT.5:

If the initial and final representations are the same, then no conversion needs to be performed.

(In effect, this rule removes a representation conversion after it has been introduced. Using some kind of "data structure history", as suggested above, such an unnecessary conversion would probably not be introduced.)

There are two rules for performing the conversion when it is actually needed:

Rule CONVERT.6:

The representation of a collection may be converted by duplicating it.

Rule CONVERT.7:

The representation of a mapping may be converted by duplicating it.

Both of these rules refine the conversion operation into one of the "duplicating" operations discussed earlier (**DUPLICATED-COLLECTION, DUPLICATED-MAPPING**).

6.6. Control structures

PECOS's specification language includes four basic control structures:

COMPOSITE
A partially-ordered set of actions to be performed[58].

LOOP
An initial action, a loop body, and a set of event indicators (similar to those suggested by Zahn [Zahn 1974]) to handle exit conditions.

TEST
A test to be performed and actions to be performed if the test succeeds and if it fails.

CASE
A set of <condition, action> pairs: each condition is a test, and if it succeeds the associated action is to be performed. The conditions must be both mutually exclusive and mutually exhaustive: exactly one of them must hold whenever a **CASE** statement is entered.

In addition, one other control structure was introduced with the rules for enumeration:

PRETEST-LOOP
A loop with a single exit test to be performed before the loop body is executed on each iteration[59].

Control structures (CONTROL)

A single rule is needed to refine a pre-test loop into the standard loop structure:

Rule CONTROL.1:
A loop with a single exit test E to be performed before the loop body B may be implemented as a loop whose body is a test whose condition is E, whose "true" action is to exit the loop, and whose "false" action is B.

The other structures are all low-level enough that there is a direct translation into LISP, so there is really no need for LISP-independent rules. Nonetheless, several interesting issues have arisen, and they will be discussed here.

----

[58] PECOS does not currently make use of the partial ordering and assumes that the actions are to be performed in the order that they are given.

[59] (a "while" loop)

6.6.1. Local memory

Both loops and composites allow for the possibility of having "local" memory: a list of LOCAL-MEMORY-UNITS with associated data structures. Each such unit is available within the scope of the control structure, and the results of particular computations may be associated with a memory unit and later retrieved from that memory unit. For most purposes, they can simply be considered to be local declarations. The one difference is that LOCAL-MEMORY-UNITS are intended to allow for several different ways of associating and retrieving values. In addition to storing the result of a computation by assigning it to a variable, one might prefer to recompute the result (e.g., if it is particularly easy to compute).

<u>Local memory (MEMORY)</u>

Unfortunately, PECOS's rules cover only the "variable assignment" technique:

<u>Rule MEMORY.1</u>:

One technique for remembering the result of a computation is to save it as the value of a variable.

It is also necessary to select the variable name to be used:

<u>Rule MEMORY.2</u>:

If a result is being saved as the value of a variable, one way to select a variable name is to invent one.

The action part of this rule includes a call to LISP's GENSYM function. Since this is PECOS's only rule for selecting variable names, the names appearing in PECOS's programs are not particularly mnemonic.

The rules above are used to select a technique for storing and retrieving values. The following two rules reflect the use of this technique in the code of the constructed program:

<u>Rule MEMORY.3</u>:

If the memory scheme of a local memory unit is to use the value of a variable named V, a value may be stored by assigning the value to the variable V.

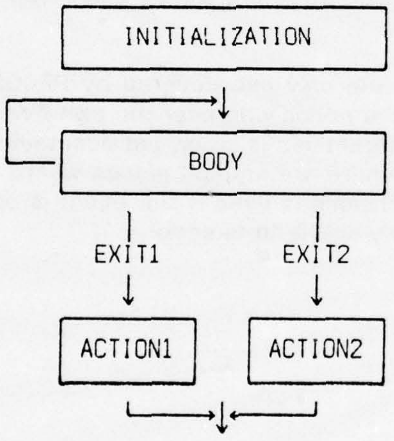
<u>Rule MEMORY.4</u>:

If the memory scheme of a local memory unit is to use the value of a variable named V, a value may be retrieved by retrieving the value of the variable V.

These two rules are actually rather obvious once the memory scheme and the variable name have been chosen. In fact, the whole issue may seem somewhat overblown, especially since rules for other memory techniques (such as "recompute") are not included. Ultimately, better methods for modeling data flow between computational units will have to be developed. The utility of different techniques for implementing that flow could then be more meaningfully assessed.

6.6.2. Loop exits

The loop construct employed in PECOS's rules involves the use of event indicators. Associated with each loop is a set of exit labels and associated actions. During the execution of any loop, an action may signal one of these events (using a special operation called **ASSERT-EXIT-CONDITION**). When such an event is signalled, the associated action is executed and the loop exited. Thus, the flow of control in a loop with two exit conditions can be diagrammed as shown below:



Exiting a loop (EXITS)

There are several ways that such exits can be implemented. PECOS can deal with only one of them (in a fashion similar to that used for local memory units):

Rule EXITS.1:

> ***A technique for exiting from a loop is to transfer control to a separate place where the exit action will be executed.***

This technique typically requires a label to which control can be transferred:

<u>Rule EXITS.2</u>:

> ***If the exit technique is to transfer control to a separate place and a label is needed, invent one.***

(This rule is similar to the variable name rule, and also calls the LISP function GENSYM.)

Once the exit technique and label have been selected, the occurrences of **ASSERT-EXIT-CONDITIONs** can be refined:

<u>Rule EXITS.3</u>:

> ***If the exit technique is to transfer control to a separate place whose label is L, an operation of asserting the event condition may be implemented by a transfer of control to the label.***

The actual implementation of the loop structure will be discussed in connection with the LISP-specific rules.

The "transfer" exit technique is the only one covered by PECOS's rules. Another common technique is to execute the action wherever the exit event is signaled. The difference between these two alternatives is minor, but nonetheless real. Typically, the transfer technique is used if there are several places where the same event is signaled, while the "in-place" technique is used if the event is only signaled in one place or the exit action is relatively simple to execute.

6.7. LISP as a target language

In order for any automatic programming system to function, it must know something about the target language, the language in which it is supposed to write programs. In PECOS's case, this language is a LISP dialect known as INTERLISP [**Teitelman 1975**]. About one-fourth of PECOS's rules deal directly with INTERLISP (hereafter referred to simply as LISP)[60]. Most of the LISP rules are quite straightforward, merely stating that specific actions can be performed by specific LISP functions. They are all expressed in exactly the same formalism as the rest of PECOS's rules. The significance of this lies in the fact that knowledge about LISP is associated with the uses to which the LISP constructs can be put. Rather than describing the function CAR in terms of axioms or pre- and post-conditions, as is done in most automatic programming systems, PECOS has rules dealing with specific uses of CAR, such as returning the item stored in a cell of a "LISP list" or returning the object stored in one of the fields of a record structure represented as a CONS cell. Thus, there is never a necessity of searching through the knowledge base of facts about LISP in order to see whether some function will achieve some desired result. In effect, that information is stored with the description of the result. The effect of this representation is two-fold: searching is significantly reduced, but so are the possibilities of "inventing" some new use for a particular LISP function.

LISP code for syntactic primitives (LISPSYNTAX)

Several rules deal primarily with LISP syntax:

Rule LISPSYNTAX.1:

> ***In LISP, a program with name P, argument list A, and body B, is written as a function definition with name P, argument list A, and definition body B[61].***

Rule LISPSYNTAX.2:

> ***In LISP, a function definition, with argument list A and definition body B, is written as (LAMBDA A B).***

Of course, the internal form of the rule must distinguish between the "constant" LAMBDA and the "variables" A and B.

----

[60] In a preliminary experiment, these rules were replaced by rules for some aspects of SAIL, an ALGOL-like language [**Reiser 1976**]. Through the use of both the "general" rules and the SAIL rules, PECOS succeeded in writing a few simple SAIL programs.

[61] After all, the knowledge that a function definition is a program has to exist somewhere!

<u>Rule LISPSYNTAX.3</u>:

In LISP, a call to the function named F, with argument expressions A_1, A_2, ..., A_n, is written as (F A_1 A_2 ... A_n).

<u>Rule LISPSYNTAX.4</u>:

In LISP, the evaluation of a variable named X is written as X.

<u>Rule LISPSYNTAX.5</u>:

In LISP, an atomic constant with value V is written as (QUOTE V).

<u>Rule LISPSYNTAX.6</u>:

In LISP, an integer constant with value V is written as V.

<u>Rule LISPSYNTAX.7</u>:

In LISP, a string constant with value V is written as V.

<u>Rule LISPSYNTAX.8</u>:

In LISP, the Boolean constant with value "True" is written as T.

<u>Rule LISPSYNTAX.9</u>:

In LISP, the Boolean constant with value "False" is written as NIL.

Rules are needed for a few of the basic functions:

<u>Rule LISPSYNTAX.10</u>:

In LISP, an assignment of a value V to a variable X may be implemented with a call to the function SETQ with X and V as its arguments.

<u>Rule LISPSYNTAX.11</u>:

In LISP, a test of whether two structured objects are the same may be implemented as a call to the function EQUAL with the two objects as its arguments.

From this point on, the phrase "In LISP" will be omitted from the rule descriptions.

LISP control structures (LISPCONTROL)

LISP has only a few, relatively simple, control structures, but they can be used fairly easily to implement the control structures of PECOS's specification language. While developing rules to describe this process, an attempt was made to break the process down into its most primitive steps. In retrospect, this attempt was probably misguided. Although it may indeed have succeeded, it was probably not worth the effort. In the long run, a more reasonable solution to the process of implementing such simple control structures as **COMPOSITE**s and **LOOP**s would be to have it done as a post-process, after all of the other parts of the program have been implemented. But before that can be done, better methods of describing control flow must be developed. In the interests of clarity, the detailed rules that PECOS uses will be omitted and a considerably simplified version presented.

Rule LISPCONTROL.1:

A composite, with parts P1, P2, ... Pn, and with no local memory may be implemented as a call to the function PROGN.

The LISP statement then has the following form:

```
(PROGN P1 P2 ... Pn)
```

Rule LISPCONTROL.2:

A composite, with parts P1, P2, ... Pm, and local memory units M1, M2, ... Mn, may be implemented as a call to the function PROG; if Pm returns a value, then use (RETURN Pm) in place of Pm.

This rule produces code of the following form:

```
(PROG (M1 M2 ... Mn)  P1 P2 ... Pm)
```

Rule LISPCONTROL.3:

A loop with initialization I, body B, exit event indicators E1, E2, ... Ek, and local memory units M1, M2, ... Mn, may be implemented as a call to the function PROG.

The following PROG structure is produced:

```
(PROG (M1 M2 ... Mn)
      I
 L1   B (GO L1)
 E1   E1/action (GO L2)
 E2   E2/action (GO L2)
 ...
 Ek   Ek/action (GO L2)
 L2)
```

In addition, the function RETURN must be used wherever appropriate (e.g., to return a value from the loop).

Rule LISPCONTROL.4:
A test with condition C, "True action" TA, and "False action" FA, may be implemented as a call to the function COND.

The following code is produced:

```
(COND (C TA)
      (T FA))
```

Rule LISPCONTROL.5:
A test with condition C and "True action" TA, but no "False action", may be implemented as a call to the function COND.

The following code results:

```
(COND (C TA))
```

Rule LISPCONTROL.6:
A case with pairs <C1, A1>, <C2, A2>, ... <Ck, Ak>, may be implemented as a call to the function COND.

This rule results in the following code:

```
(COND (C1 A1)
      (C2 A2)
      ...
      (Ck Ak))
```

LISP input and output (LISPIO)

Most input and output in LISP is done with respect to single expressions, either reading one in or printing one out.

Rule LISPIO.1:
A single datum may be input by calling the function READ with no arguments.

Rule LISPIO.2:

A single datum may be output by calling the function PRINT with the datum as its argument.

Rule LISPIO.3:

A string message may be printed on the teletype by calling the function PRIN1 with the string as its argument.

Rule LISPIO.4:

An end-of-line may be printed on the teletype by calling the function TERPRI with no arguments.

There is also a rule for a special case in which the data structure being output is almost, but not quite, the same as the output representation:

Rule LISPIO.5:

If a collection C represented as a LISP list with a special header cell, it may be output as a LISP list without a special header cell by calling the function PRINT, with its argument being a call to the function CDR with C as its argument.

The knowledge embedded in this rule belongs more properly with the rules about converting between representations for collections.

LISP lists of CONS cells (LISPLIST)

In the summary of "linked free cells" given earlier, several data structures and operations were given. All of those map relatively directly into particular LISP data structures and functions.

Rule LISPLIST.1:

Linked free cells may be represented as a LISP list of CONS cells.

Rule LISPLIST.2:

A test of whether an item is stored in some linked free cell of a LISP list may be implemented as a call to the function MEMBER with the item and list as its arguments.

Rule LISPLIST.3:

If linked free cells are represented as a LISP list, a link to a new cell, with item X and link L, may be created by creating a new CONS cell with car-part X and cdr-part L.

Rule LISPLIST.4:

A new CONS cell with car-part A and cdr-part D may be created by calling the function CONS with argument list (A D).

Rule LISPLIST.5:

If linked free cells are represented as a LISP list, a retrieval of the link from a cell C may be implemented by retrieving the cdr-part of C.

Rule LISPLIST.6:

The cdr-part of a CONS cell C may be retrieved by calling the function CDR with C as its argument.

Rule LISPLIST.7:

If linked free cells are represented as a LISP list, a retrieval of the item from a cell C may be implemented by retrieving the car-part of C.

Rule LISPLIST.8:

The car-part of a CONS cell C may be retrieved by calling the function CAR with C as its argument.

Rule LISPLIST.9:

If linked free cells are represented as a LISP list, the link of a cell C may be replaced by a new link L by replacing the cdr-part of C with L.

Rule LISPLIST.10:

The cdr-part of a CONS cell C may be replaced with a new value V by calling the function RPLACD with C and V as its argument.

Rule LISPLIST.11:

If linked free cells are represented as a LISP list, the item of a cell C may be replaced by a new item L by replacing the car-part of C with L.

Rule LISPLIST.12:

The car-part of a CONS cell C may be replaced with a new value V by calling the function RPLACA with C and V as its argument.

Rule LISPLIST.13:

If linked free cells are represented as a LISP list, a new instance of the empty link may be created by creating a new instance of the LISP atom NIL.

Rule LISPLIST.14:

If linked free cells are represented as a LISP list, a test of whether a link is the empty link may be implemented as a call to the function NULL with the link as its argument.

Finally, there is a rule for determining whether two collections have matching representations (c.f., the discussion about REPRESENTATION-MATCH queries in section 4.3):

Rule LISPLIST.15:

If two collections are represented as LISP lists and the representations of their elements match, then the two collections match.

(There are also clauses to check special header cells and whether or not both lists are ordered.)

There are similar representation match rules for other LISP data structures, and they will be omitted in the interest of brevity.

Plexes represented with CONS cells (LISPCONS)

One of the data structures in PECOS's specification language is an abstract record structure called a PLEX. PLEXs can be implemented using CONS cells, but care must be taken when the plex has more than two fields.

Rule LISPCONS.1:

A plex with two parts, A and B, may be represented as a CONS cell whose car-part is A and whose cdr-part is B.

Rule LISPCONS.2:

> ***A plex with more than two parts (A being the first and B being a list of the rest) may be represented as a CONS cell whose car-part is A and whose cdr-part is a plex with whose parts are those in the list B.***

Note that, in effect, LISPCONS.2 is a recursive rule, enabling plexes with many fields to be built up out of the two field available in a CONS cell. In the rest of this discussion, only the rules for plexes with two fields will be given. The rules for plexes with more than two parts are similar.

Rule LISPCONS.3:

> ***If a plex is represented as a CONS cell, a new instance of the plex, with first field F1 and second field F2, may be created by creating a new CONS cell with car-part F1 and cdr-part F2.***

The internal form of the rule is a little more complicated in order to insure that the correct field is used for the car-part and the correct field foı the cdr-part. Also note that the CONS function rule given earlier with the LISPLIST rules can now be applied.

Rule LISPCONS.4:

> ***If a plex P is represented as a CONS cell, and the field F is stored in the car-part, the object in field F may be retrieved by retrieving the car-part of P.***

Rule LISPCONS.5:

> ***If a plex P is represented as a CONS cell, and the field F is stored in the cdr-part, the object in field F may be retrieved by retrieving the cdr-part of P.***

Rule LISPCONS.6:

> ***If a plex P is represented as a CONS cell, and the field F is stored in the car-part, the object in field F may be replaced by replacing the car-part of P.***

Rule LISPCONS.7:

> ***If a plex P is represented as a CONS cell, and the field F is stored in the cdr-part, the object in field F may be replaced by replacing the cdr-part of P.***

After each of these rules, one of the LISPLIST rules given earlier may be applied to produce the correct function calls.

LISP arrays (LISPARRAY)

INTERLISP allows arrays as a data type, and the following rules enable PECOS to take advantage of this:

Rule LISPARRAY.1:

> ***An array with maximum index M and minimum index 1 may be represented as a LISP array with size M.***

With a facility for implicit mappings (e.g., mapping X into X+1), it would be possible to deal with arrays whose minimum value was something other than 1.

Rule LISPARRAY.2:

> ***A new instance of an array represented as a LISP array with size S may be created by calling the LISP function ARRAY with argument S.***

Rule LISPARRAY.3:

> ***The value at index I of an array A represented as a LISP array may be retrieved by calling the function ELT with A and I as its arguments.***

Rule LISPARRAY.4:

> ***A value V may be stored at index I of an array A represented as a LISP array by calling the function SETA with A, I, and V as its arguments.***

Note that INTERLISP supports dynamic allocation of array storage and the passing of array handles as variable values. Many languages do not have this flexibility. In order to enable PECOS to deal with such languages, some of the "general" array rules will probably have to be changed. This is the most glaring example of the way that hidden assumptions about the target language can find their way into supposedly language-independent rules.

LISP hash arrays (LISPHASH)

INTERLISP also has convenient functions for creating hash tables to associate values with arbitrary pointers. PECOS's knowledge of these functions is its sole knowledge about hash tables (i.e., PECOS really doesn't know very much about hashing).

Rule LISPHASH.1:

> ***A hash table whose keys are unique pointers and whose default value is NIL may be represented as a LISP hash array.***

The rule also specifies that the size of the array will be 100. The appropriate size, of course, depends on what is known about the distribution of the keys.

Rule LISPHASH.2:

A new instance of a hash table of size S may be created by calling the function HARRAY with argument S.

Rule LISPHASH.3:

The value for key K of a hash table H represented as a LISP hash array may be retrieved by calling the function GETHASH with K and H as its arguments.

Rule LISPHASH.4:

A value V may be stored with key K of a hash table represented as a LISP hash array H by calling the function PUTHASH with K, V, and H as its arguments.

Property lists (LISPPROP)

In most LISP systems, every atomic object has an associated structure known as a property list. Property lists provide a way of associating named values with atoms. One of the more interesting uses of property lists is to implement distributed mappings:

Rule LISPPROP.1:

A distributed mapping whose domain elements are LISP atoms and whose default image is NIL may be represented as LISP property list values.

The default image must be NIL, since NIL is returned when attempting to retrieve the property value of an atom that does not have the property.

As mentioned in connection with the rules about mappings (section 6.4), distributed structures cannot be passed as values of computations. Instead, some way of accessing the distributed parts must be passed. For property list values, the property name can be passed. Thus, if a collection is represented as a distributed Boolean mapping, which in turn is represented with property list values for the property XYZ, the string XYZ is precisely what is passed around as the handle of the collection. Thus, when initializing such a collection, a new name must be invented:

Rule LISPPROP.2:

A new instance of a distributed mapping represented as LISP property values may be created by inventing a new property name.

Rule LISPPROP.3:
A new property name may be invented by a call to the function GENSYM with no arguments.

In contrast with the variable name and transfer label rules, which merely used GENSYM to invent new names while the program was being constructed, at applying LISPPROP.3 results in a call to GENSYM actually appearing in the constructed program. Thus, the following code initializes such a mapping and assigns it to the variable X: (SETQ X (GENSYM)).

Rule LISPPROP.4:
If a distributed mapping M is represented as property list values, the retrieval of the image of a domain element D may be implemented as the retrieval of the property named M from the atom D.

Rule LISPPROP.5:
The retrieval of the property named P from the atom A may be implemented as a call to the function GETPROP with arguments A and P.

Rule LISPPROP.6:
If a distributed mapping M is represented as property list values, a value V may be stored as the image of a domain element D by putting V as the property named M on the atom D.

Rule LISPPROP.7:
Putting a value V as the property named P on an atom A may be implemented as a call to the function PUTPROP with arguments A, P, and V.

LISP atoms (LISPATOM)

Rule LISPATOM.1:
A string primitive may be represented as a LISP atom whose print name is the string.

Rule LISPATOM.2:
A new instance of a string primitive represented as a LISP atom with print name P may be created by using an atomic constant with value P.

Rule LISPATOM.3:

A test of whether an object is an atom with a particular value V may be implemented as a call to the function EQ with the object and the atomic constant V as its arguments.

The following rule is used in determining whether a hash array can be used to implement a mapping. See the earlier section on hash table rules.

Rule LISPATOM.4:

A LISP atom is a unique pointer.

LISP integers (LISPINT)

Rule LISPINT.1:

An integer primitive may be represented as a LISP integer.

Rule LISPINT.2:

A new instance of an integer K represented as a LISP integer may be created by using an integer constant with value K.

INTERLISP distinguishes between "small" and "large" integers, with small integers being guaranteed to be unique pointers, similar to the way that atom pointers are guaranteed to be unique. The following rule enables PECOS to use this feature to construct hash tables whose keys are integers. This can be of particular value when the keys are relatively sparse in the range of possible values.

Rule LISPINT.3:

If the maximum value of an integer is less than 1536 and the minimum value is greater than -1536, an integer represented as a LISP integer is a unique pointer.

PECOS also knows about several elementary integer functions:

Rule LISPINT.4:

A test of whether an integer I is greater than an integer J may be implemented as a call to the function IGREATERP with I and J as its arguments.

<u>Rule LISPINT.5</u>:

The sum of two integers, I and J, may be computed by calling the function IPLUS with I and J as its arguments.

<u>Rule LISPINT.6</u>:

The difference between two integers, I and J, may be computed by calling the function IDIFFERENCE with I and J as its arguments.

<u>Rule LISPINT.7</u>:

The product of two integers, I and J, may be computed by calling the function ITIMES with I and J as its arguments.

LISP Booleans (LISPBOOL)

<u>Rule LISPBOOL.1</u>:

A Boolean primitive may be represented as a LISP Boolean.

<u>Rule LISPBOOL.2</u>:

A new instance of a LISP Boolean with value V may be created by using a Boolean constant with value V.

<u>Rule LISPBOOL.3</u>:

A LISP Boolean with value "False" is a pointer to NIL.

This is the rule used in checking the default images for arrays and hash tables.

<u>Rule LISPBOOL.4</u>:

The negation of a predicate may be computed by calling the function NOT with the predicate as its argument.

<u>Rule LISPBOOL.5</u>:

A test of whether any test (in a list of tests) is true may be implemented as a call to the function OR with the list of tests as its arguments.

Rule LISPBOOL.6:

A test of whether every test (in a list of tests) is true may be implemented as a call to the function AND with the list of tests as its arguments.

6.8. Index of rule topics

7. SAMPLE PROGRAMS

As an indication of the range of PECOS's capabilities, several sample programs will be presented in this section[62]. The first four of these were selected as target programs early in the research, in order to have a focus for the development of the rules. After most of the rules were written, the last two were selected as a way of testing the generality of the rules.

7.1. Membership test

The variety of implementations that PECOS can produce is illustrated well by a simple membership test specification:

```
DATA STRUCTURES
         Y            a collection of integers
         X            an integer

ALGORITHM
         is X an element of Y
```

PECOS can implement such a test in about a dozen ways, differing primarily in the way that Y is represented. If Y is represented as a sequential collection, there are several possibilities. In the special case of a linked list, the LISP function MEMBER can be used. In addition, there are various ways of searching that are applicable for either linked lists or arrays. If the collection is ordered, the search can be terminated early when an element larger than X is found. If the collection is unordered, the enumeration must run to completion. A rather strange case is an ordered enumeration of an unordered collection, whose time requirement is order n^2. If Y is represented as a Boolean mapping, a membership test is implemented as the retrieval of the image of X. For each way to represent a mapping, there is a way to retrieve the image of X. The LISP functions GETHASH, GETPROP, and ELT apply to hash arrays, property list markings, and arrays respectively. In addition, a collection of pairs can be searched for the entry whose CAR is X and return its CDR. PECOS has successfully implemented all of these cases.

----

62 Theoretically PECOS can implement any algorithm that can be described in its specification language. In practice, however, PECOS cannot handle specifications much longer than "a page" before space limitations become prohibitive.

7.2. A simple classification program

The second target program was a simple classification program called CLASS. CLASS inputs a set (called the "concept") and then repeatedly inputs other sets (called "scenes") and classifies them on the basis of whether or not the scene fits the concept. A scene fits a concept if every member of the concept is a member of the scene. The specification given to PECOS is paraphrased below[63]:

```
DATA STRUCTURES
          CONCEPT          a collection of integers
          SCENE            a collection of integers

ALGORITHM
          CONCEPT ← input a list of integers;
          loop:
                    SCENE ← input a list of integers or the string "QUIT";
                    if SCENE = "QUIT" then exit the loop;
                    if CONCEPT is a subset of SCENE
                          then output the message "Fit"
                          else output the message "Didn't fit";
                    repeat;
```

The major variations in implementations of CLASS involve different representations for SCENE and the role they play in the subset test. The test is refined into an enumeration of the elements of CONCEPT, searching for one that is not a member of SCENE[64]. In the simplest case, the internal representation of SCENE is the same as the input representation, a linked list. The other cases involve converting SCENE into some other representation before performing the subset test. The major motivation for such a conversion is that membership tests for other representations are much faster than for an unordered linked list. One possibility is to sort the list, but the time savings in the membership test may not be sufficient to offset the time required to perform the sorting[65]. Other possibilities include the use of Boolean mappings such as property list markings and hash tables. PECOS has successfully constructed all of these variations.

----

63 Integers were used as the elements of the scenes and concept to facilitate the use of ordered collections. A different set of implementations would be possible with different types of elements in the sets.

64 For some representations that PECOS cannot yet handle, other forms for the subset test are appropriate. For example, if CONCEPT and SCENE are both represented as bit vectors, "SCENE v ¬CONCEPT" is non-zero if and only if CONCEPT is a subset of SCENE.

65 PECOS cannot currently use the technique of sorting both lists so that they can be scanned in parallel, thereby greatly increasing the savings.

7.3. A simple concept formation program

The third target program was TF, a rather simplified version of Winston's concept formation program [Winston 1975]. TF builds up an internal model of a concept by repeatedly reading in scenes which may or may not be instances of the concept. For each instance, TF determines whether the scene fits the internal model of the concept and verifies the result with the user. The internal model is then updated based on whether or not the result was correct. The internal model consists of a set of relations, each marked as being "necessary" or "possible". A scene fits the model if all of the "necessary" relations are in the instance. The full algorithm, including the updating process, is given below:

DATA STRUCTURES

RELATION	a relation name and a list of arguments
CONCEPT	a mapping of RELATIONs to {"NECESSARY", "POSSIBLE"}
SCENE	a collection of RELATIONs
TEST-RESULT	a Boolean
USER-RESPONSE	either "CORRECT" or "WRONG"

ALGORITHM

```
loop:
    SCENE ← input a list of relations or the string "QUIT";
    if SCENE = "QUIT" then exit the loop;
    if CONCEPT⁻¹["NECESSARY"] is a subset of SCENE
        then TEST-RESULT ← "True"
        else TEST-RESULT ← "False";
    if TEST-RESULT
        then output the message "Fit"
        else output the message "Didn't fit";
    USER-RESPONSE ← input a string;
    if TEST-RESULT ∧ USER-RESPONSE="CORRECT"
        then for all R in SCENE:
            if R is not in the domain of CONCEPT
                then CONCEPT[R]←"POSSIBLE"
    else if TEST-RESULT ∧ USER-RESPONSE="WRONG"
        then for any R in CONCEPT⁻¹["POSSIBLE"]:
            CONCEPT[R] ← "NECESSARY"
    else if ¬TEST-RESULT ∧ USER-RESPONSE="CORRECT"
        then nothing
    else if ¬TEST-RESULT ∧ USER-RESPONSE="WRONG"
        then for all R in CONCEPT⁻¹["NECESSARY"]:
            CONCEPT[R] ← "POSSIBLE";
    repeat.
```

The most interesting variations in the implementation revolve around the representation of the mapping CONCEPT. The uses of this mapping are similar to those of the MARKS mapping in the Reachability Program, and an inverted mapping is also appropriate here. In this case, there are two sets, NECESSARY and POSSIBLE. Since "any" and "all" operations are applied to these sets, a stored collection is appropriate (although for some distributions of input data Boolean mapping representations may be better). Since elements will be added and removed from both sets, linked lists are reasonable representations. The computation of the "domain" of CONCEPT is fairly interesting, since the domain set does not exist explicitly with inverted mappings, but must be computed, in this case by a union of NECESSARY and POSSIBLE. Note, however, that the only operation applied to the domain is a membership test. In such a case, the test can be refined into an "or" of two membership tests, one on NECESSARY and one on POSSIBLE, and there is no need to explicitly compute the domain of CONCEPT. This is the implementation that PECOS constructed.

7.4. Sorting

PECOS's development originally began as an investigation into the programming knowledge involved in simple sorting programs **[Green and Barstow 1975, 1977a, 1977b]**. PECOS's current rule set is sufficient to construct selection and insertion sorts within the transfer paradigm (transferring elements from one collection to another, such that the final collection is ordered), using both arrays and lists for the input and output collections.

7.5. Reachability

The Reachability Problem (discussed in section 2) was selected as a way of testing the generality of the rules after PECOS had already handled the above programs successfully. Very few additional rules were required to enable PECOS to implement it. The variations that have been successfully constructed involve different representations for the EXPANDED, BOUNDARY, and UNEXPLORED sets and for the SUCCESSORS mapping.

7.6. Primes

Exercise 7.1-32 of Chapter 7 of Knuth's textbook series describes the following algorithm (attributed to R. Gale and V. R. Pratt) for computing all of the nonprime odd integers less than a given integer **[Knuth 1977]**:

DATA STRUCTURES

C	a set of integers
S	a set of integers
J	an integer
N	an integer

ALGORITHM

```
N ← input an integer;
J ← 3;
C ← {1};
S ← {1};
loop:
     if J≥N/3 then exit;
     if J is not a member of C then
          for all X in S:
               if J*X<N
                    then add J*X to S;
                         add J*X to C;
                    else remove X from S;
               "Repeat until all elements of S have been handled,
                    including those which were just newly inserted";
          remove J from C;
     J ← J+2;
     repeat;
"Now C contains all the nonprime odd numbers less than N."
```

In this algorithm, J is stepped through the odd primes and multiplied with each element of S in order to determine the integers to add to C, the set of nonprimes. Integers are added to and removed from S so that each nonprime will be added to C only once (when its greatest prime factor is J). Since each integer is added to and removed from S at most once, the number of set operations on S (as well as on C) is *O(n)*. Since each step in the algorithm includes one of these set operations, the total running time of the algorithm is *O(n)* if the set operations can be done in linear time[66].

As described above, the algorithm includes an enumeration over a collection (S) that is being modified during the enumeration process. PECOS's rules cannot yet handle such enumerations, so the algorithm was changed slightly by splitting S into two sets, S1 and S2. In addition, an algorithm for computing the primes (the collection P) from the nonprimes was added. The specification given to PECOS is paraphrased below:

----

[66] Several other linear time prime algorithms have also been developed recently [Gries 1977, Mairson 1977].

DATA STRUCTURES

C	a set of integers
P	a set of integers
S1	a set of integers
S2	a set of integers
J	an integer
K	an integer
N	an integer

ALGORITHM

```
    N ← input an integer;
    J ← 3;
    C ← {1};
    S1 ← {1};
    loop:
        if 3*J≥N then exit;
        if J is not a member of C then
            S2 ← S1;
            S1 ← {};
            loop until S2 is empty:
                for any X in S2:
                    remove X from S2;
                    if J*X<N then
                        add X to S1;
                        add J*X to S2;
                        add J*X to C;
            remove J from C;
        J ← J+2;
        repeat;
    K ← 3;
    P ← {};
    loop:
        if N<K then exit;
        if K is not a member of C
            then add K to P;
        K ← K+2;
        repeat;
    output P as a linked list.
```

As noted above, the efficiency of this algorithm depends strongly on the representation of the sets S1, S2, and C. The only operations being performed on S2 are addition, removal, and taking "any" element. The "any" operation suggests that a Boolean mapping may be inappropriate and the frequent destructive operations suggest that an array may be relatively expensive. Thus, an unordered linked list is a reasonable selection. Since the value of S1 is assigned to S2, a representation conversion can be avoided by using the same representation for both sets. This is especially useful in this case, since the only operation applied to S1, the addition of elements, is relatively simple with unordered linked lists. The only operations applied to C are addition and two membership tests. Such operations are fairly fast with

Boolean mappings. Since the domain elements of the mapping are integers with a relatively high density in their range of possible values, an array of Boolean values is a reasonable representation of C. PECOS has implemented the Primes Program in this way, as well as with a linked list representation for C. To check the relative efficiency of the two implementations, each was timed for various values of N (times are given in milliseconds):

N	C as linked list	C as Boolean array
10	.05	.04
50	.28	.20
100	.63	.40
500	6.40	2.02
1000	21.21	4.08

Note the approximately linear behavior of the Boolean array case and the distinctly nonlinear behavior of the linked list case.

8. RULE GENERALITY

One of the motivations that led to the development of PECOS was a hope that a core of knowledge about programming could be isolated and codified, and that this core of knowledge would be useful when other aspects of symbolic programming (or even other programming domains) are considered. Although PECOS's rules deal with many of the fundamental aspects of symbolic programming, it is certainly far too early to say that this hope has been realized. However, several experiments, both formal and informal, have been made, with the hope of gaining a better understanding of the utility and generality of PECOS's rules.

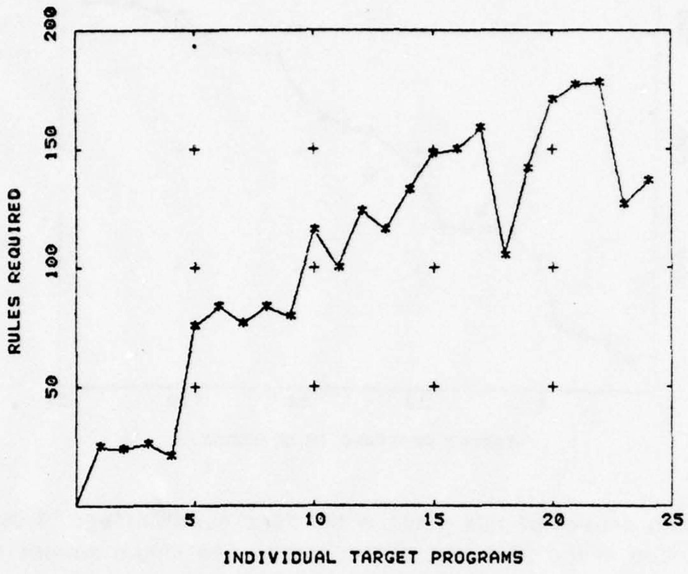
8.1. A sample set of target programs

For the purposes of these experiments, a sample of twenty-four programs has been selected. While the sample can hardly be classified as "random", it offers sufficient variety that some trends may be observed. These programs are all taken from those discussed in the previous section, and are listed in approximately the order that they were attempted while PECOS was being developed.

1. Membership: LISP function MEMBER applied to a list
2. Membership: LISP function ELT applied to a Boolean array
3. Membership: LISP function GETHASH applied to a hash table
4. Membership: LISP function GETPROP applied to property list markings
5. Membership: unordered enumeration over an unordered list
6. Membership: unordered enumeration over an unordered array
7. Membership: ordered enumeration over an ordered list
8. Membership: ordered enumeration over an ordered array
9. Membership: search through a list of <element, flag> pairs
10. Membership: ordered enumeration over an unordered list
11. CLASS: SCENE represented as a list
12. CLASS: SCENE converted to a hash table
13. CLASS: SCENE converted to property list markings
14. CLASS: SCENE converted to an ordered list
15. TF: CONCEPT inverted; NECESSARY and POSSIBLE as lists
16. SORT: selection with array input and array output
17. SORT: selection with list input and array output
18. SORT: insertion with list input and list output
19. SORT: insertion with list input and array output
20. REACH: SUCCESSORS as a list of pairs; all collections as lists
21. REACH: SUCCESSORS as a list of pairs; UNEXPLORED as a Boolean array
22. REACH: SUCCESSORS converted to B. array; UNEXPLORED as a B. array
23. PRIMES: C as a Boolean array
24. PRIMES: C as a linked list

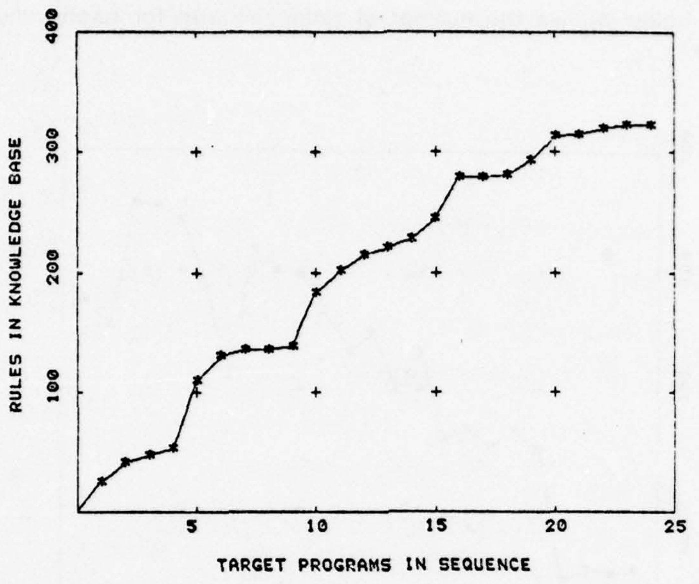
Sample Target Programs for Experiments

The chart below shows the number of rules required for each individual program in the sample.



8.2. A sequence of target programs

In a sense, the table in the previous section shows "how much programming knowledge" is required for each target program in isolation from the others. One of the major benefits of the knowledge-based approach, however, is that the targets need not be considered in isolation: each can build upon the earlier ones. Considering them in sequence, then, a growing knowledge base can be identified. In the first step, the 25 rules required for P1 are added to the knowledge base. Then the additional rules required for P2 are added. Since 8 of P2's 24 rules were also required for P1, only 16 more need be added. The graph below shows the growth of the knowledge base as each program in the sequence is achieved:



The most interesting aspect of this graph is the "scalloped" effect. This effect is especially pronounced in the first part of the graph. The shape suggests that the process of accumulating the knowledge base goes through a sequence of stages. The beginning of each stage involves the acquisition of a core of knowledge about some part of the domain, and the rest of the stage involves acquiring a little more so that the knowledge can be applied to several specific tasks. The stages in the above graph can be characterized by the aspect of symbolic programming that is involved:

Basics
The first fifty rules (for programs P1 through P4) constitute many of the basics that are needed to write programs, such as the syntax of function calls in LISP, as well as the simplest data structure refinement chains for collections and mappings.

Enumeration
The next ninety rules (for P5 through P9) constitute the knowledge necessary for writing simple enumerations. Also included are the rules for simple operations on array subregions as a representation for collections.

Ordered Enumeration
The next fifty (for P10) deal largely with ordered enumerations of unordered collections.

Destructive Operations and Inverted Mappings
The next fifty (for P11 through P15) include most of the rules for destructive operations on unordered collections and the conversion of collections and mappings from one representation to another.

Insertion into Ordered Sequential Collections

The next fifty rules (for P16 through P19) deal primarily with insertion into ordered sequential collections, as well as the few additional rules required for simple transfer sorting (most of the necessary knowledge was already in the knowledge base).

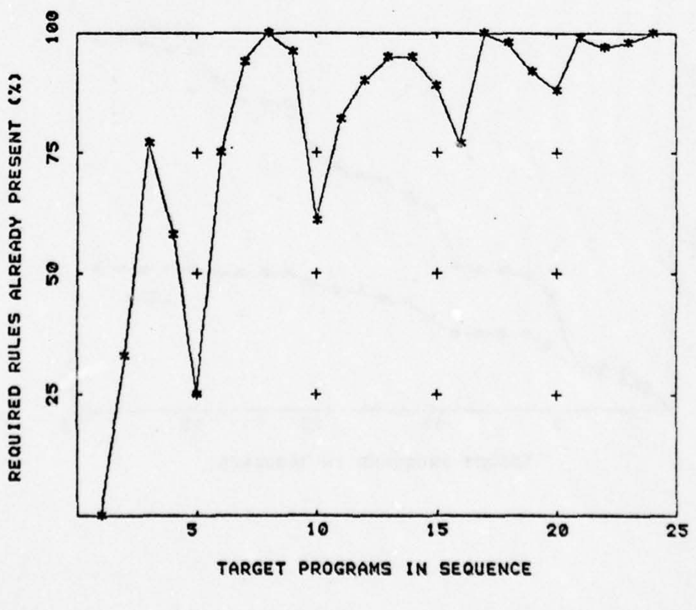
"For any"'' and Numeric Operations

The next thirty (for P20 through P24) are concerned primarily with techniques for taking "any" element of a collection. Also included are rules for a few simple numeric operations.

At each stage, of course, a few rules that might be termed "basics" were also added, but generally each stage had a dominant theme.

8.3. Toward the development of a useful core of knowledge

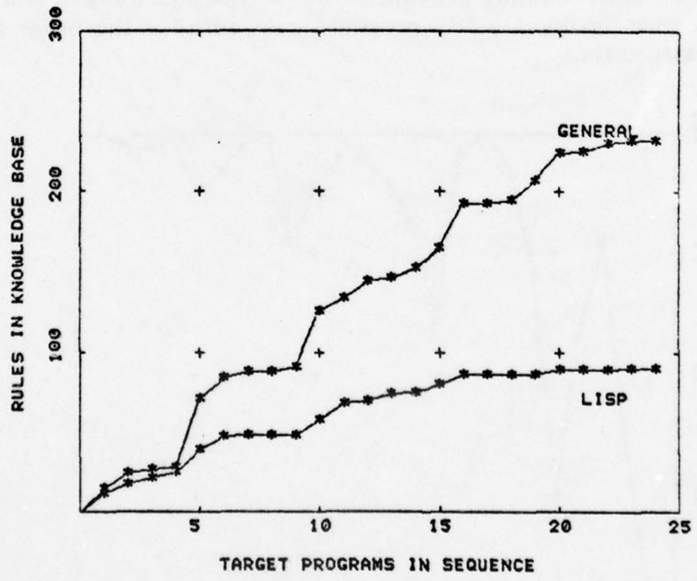
Perhaps the most interesting question revolves around the utility of the knowledge base for each successive program in the sequence. That is, given the knowledge base at some point and the next target program, how much do the already codified rules help in the achievement of that program. This question can be made more precise: at the time that a program in the sequence is achieved, what proportion of the needed rules were already present in the knowledge base? The chart below shows this data for the twenty-four programs considered in the order in which they were actually attempted:



As can be seen, the trend is toward increasingly larger amounts of the required knowledge to be in the knowledge base. While the sample set is certainly too small and non-random to justify general conclusions, such a trend is an encouraging sign that the rules embody programming knowledge with a fairly high utility in symbolic programming. Perhaps the best sign is that almost 90% of the rules required for the Reachability and Primes Programs had been written before the two programs were even considered as potential targets.

8.4. The role of the target language rules

PECOS's rules can be divided into two categories, those of a relatively general nature and those specific to LISP as a target language. Of PECOS's current 406 rules, 119 (about 30%) are LISP-specific. It is interesting to note that the ratio of general rules to LISP rules has steadily increased during the development of the knowledge base. At the beginning, it was necessary to encode large amounts of LISP specific knowledge. But once that knowledge had been put into rule form, more general kinds of knowledge could be added. The latter stages of rule development concentrated almost totally on developing and increasing the store of general rules. The two growth rates are shown in the graph below:

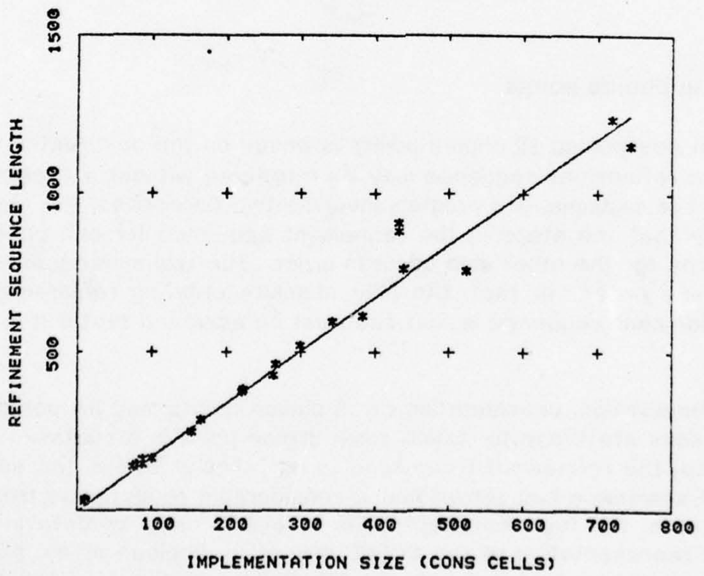


9. A SEARCH SPACE OF CORRECT PROGRAMS

As noted in section 3.2, PECOS's refinement tree constitutes a space of correct programs: each path in the tree is a refinement sequence from the abstract algorithm through several partially refined descriptions to a concrete implementation. Each leaf represents a different implementation. Given some metric (e.g., an efficiency measure) for comparing these implementations, the tree can be viewed as a space to be searched to find the "best" implementation. Techniques for comparing implementations and for searching refinement trees have not been a central part of this work, although the utility of such techniques is clear. In fact, much of PECOS's development was influenced by a desire to facilitate the process of finding good implementations. Work on LIBRA, PSI's efficiency expert, is aimed at developing useful search and comparison techniques [Kant 1977]. Even in the work on PECOS alone, however, several aspects of searching and comparison have arisen.

9.1. Refinement sequences

One interesting feature of the space is that individual refinement sequences seem to vary about linearly with the size of the final implementation. The following graph is based on the twenty-four programs used for data in the previous section:



Thus, if the correct choice can be made at each choice point without exploring any alternative paths, the cost of constructing the best implementation is roughly linear with the size of the final program.

This cost may be reduced through the use of special purpose rules to handle frequently occurring cases. Such rules could collapse an entire sequence of rule applications into a single rule. While the rules to derive the sequence must still be available for use in other situations, one can frequently get by with a "short-cut" rule. Of course, by skipping the standard sequence some option occuring partially through the sequence may not be considered at all, so the circumstances under which such a rule is appropriate must be identified carefully. PECOS currently has a few such rules for some very simple cases. The further development of techniques for using such short-cut rules effectively would be a useful line for further research.

9.2. Techniques for reducing the space

There are several techniques that may be used to reduce the size of refinement trees (i.e., the total number of rule applications) without eliminating any of the alternative implementations. These techniques all deal with choice points, tasks for which more than one rule is applicable[67]. When several rules are to be applied, a refinement path is split and a separate rule applied in each branch. Thus, each branch represents a different way to achieve the choice point task. One important observation is that achieving a choice point permits reconsideration of those tasks for which it was a subtask. The achievement of these tasks may, in turn, enable higher level tasks to be considered, so the achievement of a single choice point may result in a large number of tasks being enabled and achieved.

9.2.1. Postponing choice points

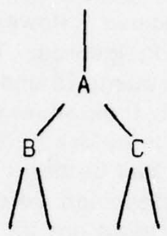
The technique of postponing all choice points is based on the observation that many of the steps in a refinement sequence may be reordered without affecting the final implementation. For example, if a program involves two collections, the only ordering requirements are that the steps in the refinement sequence for one occur in order and that the steps for the other also occur in order. The two subsequences may be intermingled in any order. In fact, the only absolute ordering requirement on two tasks in any refinement sequence is that one must be achieved first if it is a subtask of the other.

Based on this observation, consideration of all choice points may be postponed until the only other tasks are those for which some choice point is a subtask. When this technique is used, the refinement trees tend to be "skinny" at the top and "bushy" at the bottom. Experience has shown that a considerable reduction in tree size can result. For example, the four implementations of CLASS used as data in section 8 differed in their representation of the SCENE collection: Boolean array, property list marking, unordered linked list, and ordered linked list. In the tree for these four implementations, there were three choice points: (A) whether to represent SCENE as

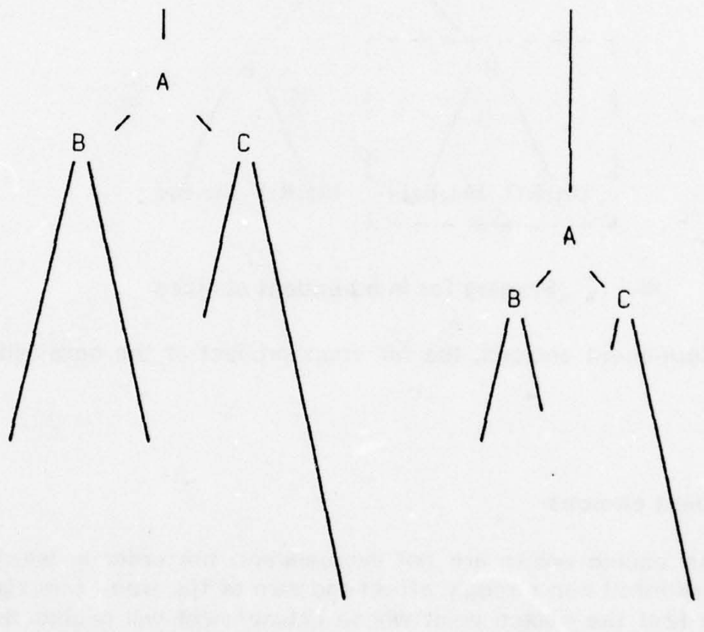
----

[67] With the current rule set, choice points are relatively infrequent. In the Reachability Program, for example, there were about three dozen choice points in a refinement sequence that involved about one thousand rule applications.

a Boolean mapping or as an explicit collection; (B) in the Boolean mapping case, whether to use a stored or distributed mapping; (C) in the explicit collection case, whether or not to keep the collection ordered[68]. The refinement tree is shown below:



When PECOS was run without postponing the choice points, the split at A occurred after 22 steps and the splits at B and C occurred after 17 and 43 more steps respectively. With the postponement technique, the splits at A, B and C were made after 236, 8, and 10 steps respectively. In the trees shown below, the path lengths are roughly proportional to the number of steps, and the difference is clear:



without postponement with postponement

Effects of postponing choice points

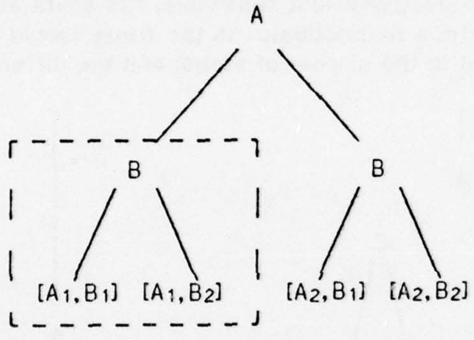
Thus, in this case, the postponement technique saved over one third of the rule applications without eliminating any of the possible implementations.

----

[68] Several other choice points were encountered, of course, but only these three are relevant to the four implementations.

9.2.2. Independent choices

When two choice points are sufficiently independent that a choice for one may be made regardless of what choice is made for the other, parts of the tree can be ignored quite easily. The leaves of the entire tree represent the cross-product of the alternatives for each of the choices. However, under the independence assumption, some of the leaves may be ignored. This may best be clarified by example. Suppose there are two choice points (A and B), each having two applicable rules. If one (say A) is considered first, then several further refinement steps (for which A had been a subtask) may be made before B needs to be considered. Since A is independent of B, the two paths for A can be carried far enough that a preference for one path over the other can be determined before B is considered. Thus, the alternatives for B along the other path need not be considered at all. In the tree shown below, the branches inside the box need not be explored if A_2 can be selected over A_1 independently of what choice is made for B.

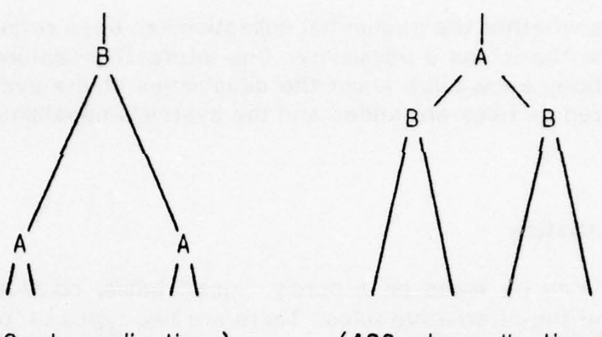


Pruning for independent choices

Thus, with independent choices, the full cross-product of the possibilities need not be explored.

9.2.3. Dependent choices

When separate choice points are not independent, the order in which the choice points are considered can strongly affect the size of the tree. One simple strategy is to consider first the choice point whose achievement will enable the refinement sequence to be carried to the greatest depth. For example, if achieving A would permit the sequence to be carried on for 10 steps while achieving B would permit carrying it on for 100 steps, the trees that result from different choice orderings would be as shown below:



(240 rule applications) (420 rule applications)

Choice ordering for dependent choices

Of course, in practice it is often hard to predict how far a refinement could be continued after applying a particular rule, but the technique is a useful guideline. The use of this technique, as well as more sophisticated ones, is being tested in connection with LIBRA [Kant 1977].

9.3. Choice making

The techniques described above help to reduce the size of the search space, but do not remove the problem of actually selecting one of the alternatives at a choice point. Intermediate-level abstractions facilitate such choice-making by focusing on the essential aspects of a choice while ignoring irrelevant details[69]. A consequence is that it is usually unnecessary to complete all implementations in order to determine which alternative is preferable. While testing rules and producing particular target programs, several choice-making heuristics, based on these intermediate-level abstractions, were added to PECOS. Such heuristics can be divided into three relatively distinct categories.

9.3.1. Heuristics for avoiding paths that fail

Although most refinement sequences succeed in producing concrete implementations, paths occasionally fail by reaching a situation in which no rules are applicable[70]. Several of PECOS's heuristics are primarily intended to recognize branches that will lead to such failure and to avoid selecting those alternatives. For example, PECOS has no rules for adding an [illegible] at the back of a linked list. One of the heuristics (for selecting a position at which an element should be added to a sequential

----

[69] The use of cost estimators for partially refined program parts is also facilitated.

[70] The cause of failure is the incompleteness of the knowledge base, not any inherent problem with the refinement path itself.

collection) tests whether the sequential collection has been refined into a linked list, and if so rejects "back" as a possibility. One interesting feature of such heuristics is that they embody knowledge about the capabilities of the system itself, and thus should be changed as rules are added and the system's capabilities change.

9.3.2. Local heuristics

Some decisions can be made on a purely "local" basis, considering only the node being refined and the alternative rules. There are two types of local heuristics:

> *One alternative is always better than another.* When one alternative is known *a priori* to be better than another, if both are applicable the better alternative should be taken. For example, one of PECOS's heuristics prefers PROGN to PROG constructs: (PROGN ...) is better than (PROG NIL ...).
>
> *It doesn't make much difference, but one is more useful.* If the cost difference between two alternatives isn't very great, but one is more convenient for most purposes, that rule should usually be chosen. For example, PECOS has a heuristic that suggests always using special headers for linked free cells, since the extra cost is low (one extra cell) and they are often more easily manipulated[71].

9.3.3. Global heuristics

Many decisions require more global considerations, including such things as the operations applied to a data structure, the relative frequency of a particular operation, and so on. None of PECOS's heuristics fit this category, but such heuristics play a major role in LIBRA [Kant 1977].

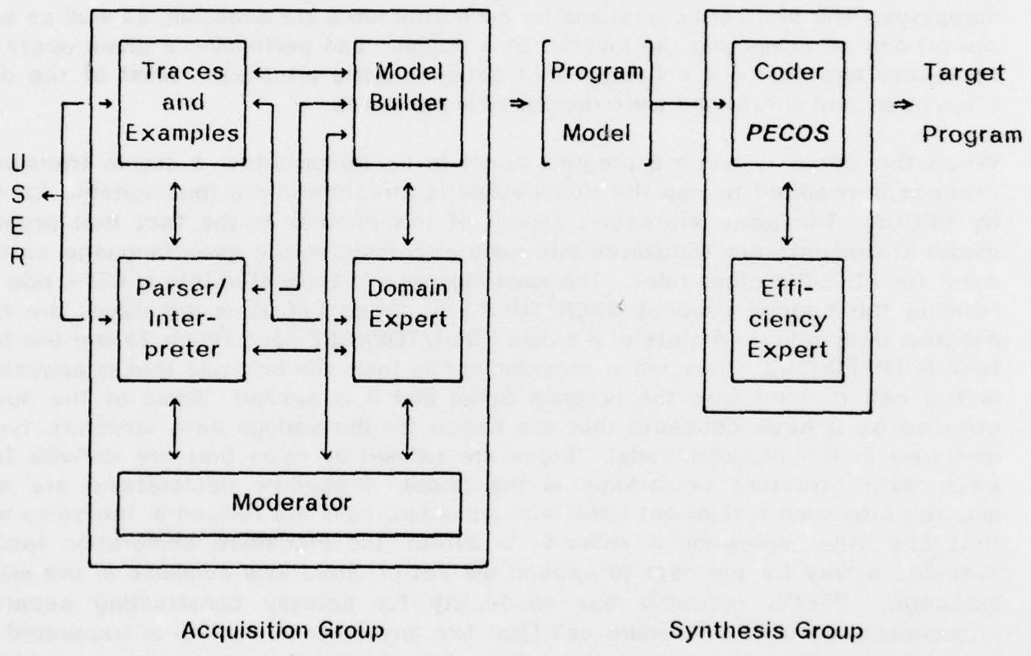
----

[71] Note that the heuristic doesn't take into account whether or not the extra cell actually helps in the particular case under consideration. And, indeed, it may be difficult to tell at the time the choice is made.

10. INTERFACE WITH THE PSI

In addition to its role as an experimental system for developing and testing rules about programming, PECOS also plays the role of the Coding Expert in the PSI program synthesis system (Ψ) being developed at the Stanford Artificial Intelligence Laboratory [Green 1976, 1977]. Ψ permits the user to specify a program through dialogue using natural language and traces. A program is then produced from this specification. For current experiments, the target programs are all from the domain of symbolic computation and the implementations of these programs are in LISP.

Ψ is a large system organized as a collection of interacting modules, as illustrated below:



Processing in Ψ occurs in two relatively distinct phases. During the *acquisition phase*, the user communicates with Ψ through either of two modules, a *parser/interpreter* [Ginsparg 1977], and a *traces and examples expert* [Phillips 1977]. The dialogue is guided by a *moderator* (developed by Louis Steinberg). The interpretation process is supported by a *domain expert*. A domain module for concept formation programs is being developed by Ronny van den Heuvel. The *model builder* [McCune 1977] assimilates the information gathered by the other modules and produces a complete and consistent description (in abstract terms) of the desired program. This description, referred to as a *program model*, is the output of the acquisition phase. During the *synthesis phase*, the *coder* and *efficiency expert* jointly determine a program that implements the program model. The basic purpose of the coder is to suggest alternative implementation techniques. The efficiency expert [Kant 1977] selects the most efficient from among the possibilities.

As the coder, PECOS interfaces primarily with two of these modules, the model builder and the efficiency expert.

10.1. Interface with the Model Builder

The program model provides the interface between the model builder and the coder. The model is described using what is essentially a high-level language for symbolic computation. Not surprisingly, this language is closely related to PECOS's specification language. The data structures available include collections and mappings. The standard operations for such structures are available, as well as such operations as computing the inverse of a mapping and performing a given operation for every member of a collection that satisfies some predicate. Most of the data structures and operations were described in section 6.

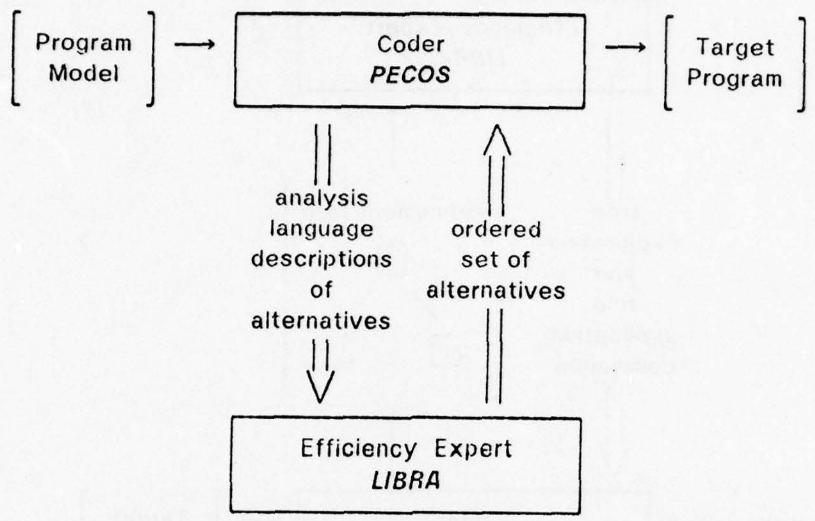
When the coder is given a program model to be implemented, a simple translation process is required to map the model builder's structure into a form suitable for use by PECOS. The most interesting aspect of this process is the fact that program model statements are translated into rules expressed in the same language as that used for PECOS's other rules. The main algorithm is translated into a REF← rule for refining the special concept USER/TOPNODE. When PECOS is initialized, the first program description consists of a single USER/TOPNODE node (node 1) and the first task is (REFINE 1). Thus, when considering this task, the only rule that is applicable is the one derived from the program model and it is applied. Some of the nodes created by it have concepts that are named for the various data structure types declared in the program model. These are refined by rules that are derived from each data structure declaration in the model. Procedure declarations are also mapped into such refinement rules, and procedure calls are refined in the same way that any other operation is refined. In effect, the procedure declaration facility provides a way for the user to expand the set of operations available in the model language. PECOS currently has no facility for actually constructing separate procedures, so each procedure call (just like any other operation) is expanded "in line," essentially in the manner of a macro. A useful extension would be the addition of a facility for writing separate procedures.

10.2. Interface with the Efficiency Expert

Two rather different paradigms for the interface between the coder (PECOS) and the efficiency expert (LIBRA) have been tried[72]. The first paradigm is illustrated below:

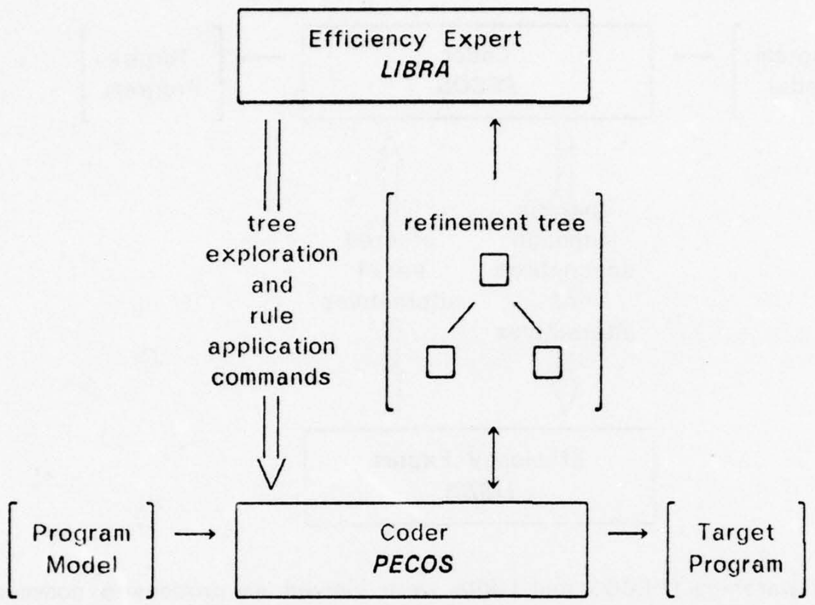
----

[72] Both were developed jointly with Elaine Kant.


Program Model
Coder
PECOS
Target Program
analysis language descriptions of alternatives
ordered set of alternatives
Efficiency Expert
LIBRA

Under this paradigm, PECOS and LIBRA were viewed as processes communicating through messages. Whenever PECOS was faced with a choice, each rule would be applied and efficiency-oriented descriptions of the results would be sent to LIBRA, which would analyze each on the basis of efficiency considerations and return a preferred ordering of the alternatives. PECOS would then continue the refinement path of the preferred alternative. Several problems were encountered with this approach. The primary problem was that the bandwidth of the message-oriented communication path was too low for efficient use. The derivation of the "efficiency-oriented" descriptions was quite expensive and the descriptions themselves were rather large. Additional problems included difficulty in determining which choice point to consider next and the necessity of embedding parts of the efficiency expert within PECOS (for example, heuristic techniques for choosing from among alternative rules without going through the relatively expensive process of applying each and analyzing the results). These considerations are discussed in more detail elsewhere [Barstow and Kant 1976].

The second (and current) paradigm is illustrated below.



Under this paradigm, the refinement tree is viewed as a search space of partially implemented programs (see section 9). By defining some efficiency measure for the implementations, the space may be searched for the leaf that minimizes that measure. PECOS provides the capability of constructing the space and LIBRA guides the exploration of that space, deciding which branches should actually be constructed and followed. Here, the refinement tree itself is the major communication mechanism, with LIBRA sending various tree exploration and rule application requests to PECOS for execution. In each cycle, LIBRA considers the refinement tree and selects a particular program description for further expansion. One of the tasks for that program description is chosen for consideration[73]. PECOS is then asked to work on that task in an appropriate fashion. Working on the task may involve retrieving and testing relevant rules, gathering bindings for a rule, or applying a rule. In addition, PECOS may be asked to split refinement paths at a given program description in the tree. In essence, LIBRA explores a tree for which PECOS is a "legal move generator." This approach allows a much higher rate of communication between the two experts, since each has access to the entire refinement tree. Note also that LIBRA is allowed considerably more freedom in determining paths to follow or suspend and tasks to consider or delay than was possible in the first paradigm. Detailed explanations of the techniques used by LIBRA to select paths and tasks are available elsewhere [Kant 1977, Barstow and Kant 1976].

----

[73] The efficiency expert uses a task structure that is similar to (but considerably more complex than) that discussed in section 5.1.

11. EXPERIENCE WITH BUILDING A RULE-BASED SYSTEM

Developing PECOS has been an experimental process. Whole sets of rules have been tried and discarded. Several different conceptual frameworks have been used with varying degrees of success. The current implementation is the third, and it certainly isn't superior to the first two in all respects. In the hope that something can be learned from the development process itself, several aspects of the process will be discussed in this section. The views presented are very subjective and based primarily on one person's experiences in developing a rule-based system for a single task. No further generality can be claimed, but discussions such as these may facilitate the development of other rule-based systems.

Two assumptions are central to this section:

> *The primary goal is the development of a system to perform a certain task.* In PECOS's case, the task was the implementation of abstract algorithms. It is important to distinguish this goal from that of "experimenting with a scheme for representing knowledge." Both are legitimate research goals, but they lead in different directions. In the task-specific case, one must keep in mind the fact that it may eventually be useful to try to apply the knowledge to a different task, and therefore guard against overly specialized representations. In the representation-oriented case, the principal danger is that the representation may not be useful in real-world applications.
>
> *It has been decided to encode domain-specific knowledge in some kind of rule form.* In PECOS's case, this decision was based on the observation that "programmers seem to know a lot about programming." The advantages and disadvantages of rule-based systems are discussed elsewhere **[Davis and King 1977]**. This section focuses on the process after such a decision has been made.

11.1. On developing a rule base

There are basically three aspects to the development of a machine-useable rule base for a specific task.

> *Explicating the knowledge.* The emphasis here is on identifying "what" is to be represented, rather than "how" to represent it. In PECOS's case, the informal "English" forms of the rules make up this stage. For most domains, such an attempt to find some kind of order or structure in the domain is a far from trivial task. Much of the work involves identifying domain-specific concepts that are often not apparent at first glance. In PECOS, for example, two key identifications were the enumeration structure (with an enumeration order and a state-saving scheme) and the notion of a sequential collection.

Establishing a conceptual framework. In contrast with the first aspect, this typically comes "in a flash of insight," when several pieces suddenly fall together. With PECOS, this happened with the recognition of the refinement tree structure.

Selecting a rule representation. Once a conceptual framework and a sample of informal rules are available, a rule representation can be designed for them. Principally, this involves recognizing certain patterns in the informal rules. In PECOS, this stage resulted in the formalism discussed in section 4.

While the development of the rule base typically involves several iterations and the three aspects cannot be totally isolated from each other, they may nonetheless be considered somewhat separately. My experience with PECOS has led me to believe that they should be considered in the order suggested above. In particular, the explication of an "informal" body of rules is a vital first step and must be carefully constructed to reflect the ideas and structures inherent in the task domain. In most domains, these ideas and structures are not known *a priori*, but must be discovered in a morass of knowledge that is available only informally or subconsciously. Since any conceptual framework or rule representation carries with it a certain perspective on the world, an attempt to use a particular framework or representation to explicate the informal rules will force the same perspective on the domain. Many domain-specific concepts or structures may not be identified or even noticed.

"Our reality is merely one of many descriptions.... My insistence on holding on to my standard version of reality rendered me almost blind to don Juan's aims.... One had to learn the new description in a total sense, for the purpose of pitting it against the old one, and in that way break the dogmatic certainty, which we all share, that the validity of our perceptions, or our reality of the world, is not to be questioned."

Journey to Ixtlan [Castaneda 1972]

In practice, this guideline is very difficult to follow. Before each implementation of PECOS, I resolved "to make a list of all the necessary rules" before designing a representation. Yet I always fell short -- in each case I thought I knew the domain well enough after a few rules (ten or twenty) that I could design a representation and express the rest of the rules formally. And I was always wrong -- in each case I later discovered kinds of rules that could not be expressed conveniently.

Another important part of rule base development involves knowing when a rule is "right" or "wrong". The answer, of course, depends on the particular task. It has been suggested that the appropriate "grain-size" of a rule is whatever human experts can view as a single chunk [Lenat and Harris 1977]. I have found two guidelines to be of particular value in the programming domain.

If a rule is very big when it is written down, it is probably wrong. There is too much going on in the rule for it all to be included in a single place.

If two rules are similar, there may be a useful underlying concept. Perhaps the common aspects can be isolated in a separate rule.

While these guidelines are certainly not absolute, they have been very helpful in developing PECOS's knowledge base.

11.2. Representation and organization of a rule base

There are five specific representation and organization techniques that greatly facilitated PECOS's development.

An agenda mechanism with interruptible tasks

Agenda mechanisms permit experimentation with a wide variety of control structures. As mentioned in section 5.1, the switch from a one-pass to a two-pass algorithm took only about an hour's work. The flexibility was increased by allowing interruption of tasks at certain specific points that did not require much of the context to be preserved. This interruptibility permitted, for example, the postponement of choice points by interrupting a task as soon as it was discovered that more than one rule was applicable.

Allowing rules to operate within rule conditions

PECOS's QUERY← rules provide a way of using rules to determine whether particular conditions hold. This helped factor the knowledge base into smaller units and facilitated adding knowledge in the form of new QUERY← rules. Without such a facility, prohibitively many rules would have required modification for each such addition. QUERY conditions can also be used as abbreviations in rule conditions. Had I realized this earlier, I would have used the QUERY facility even more than I did.

Generalized subtasks

When rules are considered and their conditions checked, there may be subtasks to be achieved. In most systems, a rule is selected and its subtasks are then considered. By contrast, PECOS generalizes the tasks slightly to avoid premature commitment to a particular rule. For example, if a rule requires that a certain data structure, X, be represented as a linked list, and X hasn't been refined yet, the standard subtask would be to "refine X into a linked list." In PECOS, this subtask is generalized to "refine X."

Separate categories of subtasks

Although PECOS's total separation of rule conditions into applicability and binding parts seems a bit ponderous, the idea of separating subtasks into different categories seems useful. It permits some tasks to be avoided completely (if their "supertasks" are achieved some other way) and other tasks to be ordered according to any kind of priority scheme.

A rule compiler

Each of PECOS's rules is automatically translated into several different forms (LISP functions and discrimination net entries) that are then used in different ways. The implementation of this rule compiler turned out to be much easier than I expected (it only took a few weeks) and it greatly increased PECOS's efficiency.

11.3. Problems and pitfalls

The biggest problem in developing PECOS's rule base was alluded to earlier: the explication of the programming knowledge itself, as opposed to the way it was represented. But there is little more that can be said about the problem except that care should be taken when constructing the informal rule list.

Another major problem with building a rule-based expert system is that there are subtle and hidden assumptions that arise from working on a particular task. For example, the operation of "creating a new instance of an array" is refined by PECOS's rules into a call to the LISP function ARRAY. The sequence of rules implicitly assumes that arrays can be allocated dynamically and array pointers passed as values of variables. This assumption didn't become apparent until rules for SAIL were attempted [Ludlow 1977]. In SAIL, arrays are allocated through declarations and cannot be passed as variable values. Before the SAIL rules are finished, this particular assumption will have to be dealt with. The problem of hidden assumptions is probably unavoidable and the best defense is simply to be aware of its existence.

Another common problem is that rules once thought correct may later be found to be applicable in incorrect circumstances. The conditions on the rules are not strict enough. This problem occurred in PECOS's development when enough rules had been added that two different refinement paths could produce similar (but not identical) structures. When this happened, rules intended to be applicable in only one were applied in both. Solving the problem required adding conditions to some of the earlier rules.

A separate problem arises when a rule should indeed be applicable in both cases (because it is "correct"), but for some reason is inferior to another that may be applicable in one (or even both). Such a preference of one correct rule over another is a good candidate for a choice-making heuristic.

Two specific problems arise when the knowledge base gets quite large. The first is that the rule-writer (or user) begins to lose his or her understanding of the knowledge base as a whole. There is no longer a coherent overview. The problem is probably unavoidable, but simple organizational techniques help to deal with it. For example, I kept the rule listings on several different files, each dealing with one general topic, and named the rules based on these topics. More sophisticated techniques may also be usefully applied. The TEIRESIAS system explored several techniques for knowledge acquisition and explanation [Davis 1976].

The other problem with large rule sets is that the rules may interact in unexpected ways. The biggest surprise in developing PECOS's rules came when PECOS considered using a hash table to represent the array allocation part of an array subregion. In retrospect, I should have guessed that this would happen, since all that was required was some kind of tabular association of elements with indices, but it was nonetheless rather unexpected. Such unexpected interactions may actually have a positive effect, since the system may then be able to deal successfully with a wider variety of situations. So this "problem" may not be a problem at all.

The final problem encountered in developing PECOS's rule set deals primarily with the rule base organization: in a very real sense the rules are not strictly independent. As the rule-writer, I am well aware of the fact that many rule conditions were written with the specific purpose of insuring that some other particular rule had been applied in a certain situation. (This occurs most frequently with operation refinement rules that are conditional on a certain data structure refinement having been made.) When that is the case, then perhaps it would be better to allow the rules to refer to each other more directly. This is, in fact, one of the extensions that should be made in PECOS's knowledge base organization -- a better facility for linking rules more closely together.

12. FUTURE DIRECTIONS

There are severally potentially fruitful research directions suggested by the results of this "experiment in knowledge-based automatic programming." Some of these involve direct extensions that could be made to the current implementation. Others involve representation and control issues that are central to the way PECOS currently operates, so it is not clear how easily they could be directly incorporated[74]. Other research directions that are suggested involve the incorporation of other techniques into the basic methodology and the application of the methodology to other tasks.

12.1. Extensions

More readable output

Most of the programs produced by PECOS are relatively hard for people to read and understand. Several extensions would help eliminate this problem. Most important is a facility for commenting the code. The history of rule applications provides a fairly complete documentation for a program and summaries of this history would be quite useful as comments in the programs themselves. Such a facility is one of the goals of Ψ's planned explanation system (being designed and implemented by Richard Gabriel). Another contribution to more "readable" output would be a post-processor that performed simple syntactic transformations. For example, (PROGN A (PROGN B) C) could be simplified into (PROGN A B C).

A constraint mechanism

Currently, rules can be applied only if all of their applicability patterns are satisfied. A useful extension would be to allow a rule to be applied in the case of an incomplete match (see section 5.3). This could be done with a mechanism for guaranteeing that the conditions would be satisfied at some later time. One technique would be to attach constraints to the subtasks returned by the incomplete match. One major benefit of permitting premature rule application is that certain branches "doomed" to failure could be avoided. Suppose, for example, that task A has a subtask B, and there are two rules for achieving B (say B_1 and B_2). However, there is only one rule for A, and it is not applicable if B_2 is applied to B. Without the constraint mechanism (or any other form of guidance), B would be considered first, causing a split in the refinement path. On the B_2 branch, the attempt to achieve A would fail. With a constraint mechanism, A could be considered first (on the general

----

[74] In fact, research in those directions is likely to suggest or confirm weaknesses that need to be corrected; it would be naive to expect that the current implementation would not require significant revision (or even a complete redesign) as the nature of these weaknesses is clarified.

principle of delaying choice points) and a constraint guarding against applying B_2 to B could be noted, thereby avoiding the B_2 branch. Such pruning can be especially significant when fairly long refinement sequences are considered. Experimentation with a constraint mechanism in an earlier implementation showed just such a pruning effect.

A pattern definition facility

There are many common constructs that can only be described rather ponderously with the current set of pattern types in the pattern matcher. For example, to specify the creation of a node for the Boolean constant "True", the following expression must be used:

```
(#NEW NEW-PRIMITIVE
      (←#RDS (#NEW PRIMITIVE
                   (←#P SPECIFIER (QUOTE BOOLEAN))
                   (←#P VALUE (QUOTE TRUE)))))
```

A simple macro facility could be used for such common constructs. For example, #TRUE could be defined as the above pattern. While it would not add to PECOS's capabilities, it would certainly simplify rule writing.

Short cut rules

Common sequences of rule applications could easily be collapsed into a single rule. PECOS currently has a few such "short cut" rules, and more could significantly increase PECOS's efficiency, as long as they didn't interfere with the exploration of useful alternatives. A very interesting possibility would be to try to acquire these short cut rules automatically on the basis of observations about the relative frequency of rule sequences.

12.2. Improvements and modifications

Procedure calls and recursion

Currently, PECOS cannot write programs with several procedures, nor can it implement any inherently recursive algorithms (such as Quicksort). The addition of a facility for writing procedure calls would be a welcome addition. One possible mechanism would be to permit an operation (at some level of abstraction) to be refined into a call to a procedure whose body was the action of some applicable refinement rule. The trick, of course, would be in determining when this would be a reasonable thing to do!

Use of state information

Information about the state of the computation is currently used in only one specific situation. When removing an element from a sequential collection, one rule tests whether the location of the element is already known. If so, the search for the location can be avoided. It is clear that much more significant use of such state information could be made. In some membership tests, for example, it may be possible to prove that the particular element can never be in the collection[75]. The element removal example was not particularly easy to express in PECOS's formalism, and it is not clear how much PECOS would have to be modified to make it more convenient to deal with state information.

Parallel representations

One interesting modification would be to permit the use of multiple, parallel representations of data structures. Currently, the only use of multiple representations involves a sequence of different representations with conversions from each representation to the next. Many interesting algorithms depend on parallel representations. For example, if a collection is represented as both a Boolean array and a linked list, both membership testing and enumeration would be fairly efficient operations. The biggest problem here is to insure that the two data structures are consistent: modifications to one must be made in parallel with modifications to the other.

Representation of data flow

Currently, the only representation of data structure usage within a program is an unordered list of operations that affect a given data structure[76]. Both of the previous two improvements would be facilitated by a more convenient mechanism for dealing with data flow. Knowing that a collection is input in one place, repeatedly modified through additions, enumerated once, and finally tested for membership several times, for example, would make it much easier to decide to use both a list and a Boolean array for that collection.

Dynamic rule development

One very interesting possibility would be the development of techniques for creating "rules" dynamically as a particular refinement progresses. Work along this line would soon get into the realm of alternative types of specification (e.g., input and output assertions instead of concept names). The problem, of course, is to figure out how

----

[75] Of course, this raises the question of resource allocation: how much effort should be expended on trying to determine whether or not the membership test will always fail?

[76] This list is maintained by PECOS primarily for use by LIBRA.

to create a rule given a specification for which no rule is applicable. In a sense, much of the classical work in automatic programming and plan development could be applied to this problem. Given some specification of an action, if it could be determined that some particular loop structure would achieve the action, a rule that refines the action into a LOOP node (with appropriate subparts) could be created. If a rule is found that "almost" fits the specifications of some action, then it could be debugged to produce a rule that is correct.

12.3. Codification of programming knowledge

One obvious extension of this work (and probably the most fruitful in the long run) is the further codification of programming knowledge. In addition to verifying (or rejecting) the utility of the knowledge-based approach, the knowledge itself could be quite useful to human programmers.

Other aspects of programming

The knowledge embodied in PECOS's rules deals with only a small part of symbolic programming. Several different representations for collections (e.g., trees) and mappings (e.g., discrimination nets and hash tables) are not covered. Many important algorithms (e.g., different kinds of searching and more efficient sorting techniques) remain to be codified. In addition, several aspects of symbolic programming are not touched by the rules at all. For example, information retrieval systems, graph algorithms, pattern matching (and even rule-based systems) are all important aspects of symbolic programming. In addition to symbolic programming, other domains that could be considered include operating systems and concurrent programs, text editors, compilers, and various types of numeric programs.

Other types of knowledge

The codification of knowledge about other tasks and domains is likely to uncover other types of programming knowledge than are apparent in PECOS's rule base. Some of the possibilities here are efficiency knowledge (such as the heuristics that play a major role in LIBRA) and temporal knowledge (such as that involved in building a structure from elements known to be arriving in a certain order). In addition, inferential knowledge is likely to grow in importance. For example, a very complex inference chain is needed to derive the standard one-pass search from a description of the desired element as the largest in a collection [Green and Barstow 1977b].

12.4. Applications of programming knowledge

Given that one has a detailed explication of programming knowledge for a particular domain, there are several applications other than program construction.

Compilers for high-level languages

In a very real sense, PECOS (with LIBRA in the synthesis phase of Ψ) is a compiler for a high-level language. The principal features of this compiler are the variety of implementations that it can produce and the ease with which more alternatives can be added. This "compiler", however, is too slow and ponderous for use as anything other than a research device. An interesting application of PECOS's knowledge base would be the development of a compiler for a "real-world" application.

Program analysis

Another interesting possibility is the use of that knowledge to "analyze" programs written by human programmers (i.e., understand the purposes of parts of the code). This line of research is being pursued by Rich and Shrobe [Rich and Shrobe 1976].

Program verification

The detailed nature of the explication should make it relatively amenable to use in automatic verification. For example, the state-saving scheme of an enumeration is closely related to the standard invariant for the enumeration loop.

Teaching

Perhaps the most intriguing application is in the area of teaching. Although it would certainly be overkill to force students to memorize the entire list of PECOS's rules, the structure imposed by the rules could be very useful in clarifying some of the issues involved in symbolic programming. For example, when running PECOS interactively, I am usually forced to consider more implementation alternatives than I would think of when programming by hand, and the necessity of making an explicit decision at each of the choice points helped me to understand the relative merits of the different representations and algorithms.

13. CONCLUSIONS

The development of PECOS represents the final stage in an experiment investigating a knowledge-based approach to automatic program construction. The essence of this approach involves the identification of concepts and decisions involved in the programming process and their codification into individual rules. These rules are then represented in a form suitable for use by an automatic programming system for application to a specific task. In PECOS's case, the particular task was that of implementing abstract algorithms in the domain of simple symbolic programming. Let us briefly review some of the results of this experiment.

As seen in section 2, the process of constructing an implementation for an abstract algorithm involves considering a large number of details. It seems a reasonable conjecture that some kind of ability to reason at a very detailed level will be required if a system is to "understand" what it is doing well enough to perform the complex tasks that will be required of future automatic programming systems. PECOS's ability to deal successfully with such details is based largely on it's access to a large store of programming knowledge.

Several aspects of PECOS's representation scheme (as discussed in sections 3, 4, and 5) contribute to this ability. First, the refinement paradigm has shown itself to be a convenient framework for coping with the complexity and variability that seem inevitable in real-world programs. The use of several levels of abstraction seems particularly important. Second, factoring the knowledge into relatively small pieces enables these pieces to be applied in a variety of different situations. The small grain size also facilitates the intelligent use of the knowledge (e.g., by a choice-making mechanism), since the pieces are more "understandable." Two techniques that have been instrumental in achieving this factoring are the use of intermediate level abstractions (such as "sequential collection") and the identification of unifying decisions (such as "state-saving scheme").

The rules presented in section 6 constitute a detailed body of knowledge about many aspects of symbolic programming. These rules deal primarily with collections and mappings and ways of manipulating such structures, including several enumeration, sorting and searching techniques. The principal representation techniques covered include the representation of sets as linked lists and arrays (both ordered and unordered), and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings. In addition to these general constructs, many low-level programming details are covered. Although the rules were developed to deal with simple symbolic programs in the domain of concept formation, they have been successfully applied in other situations as well, including algorithms for solving the reachability problem in graph theory and for generating prime numbers. The successful application of the rules in such varied domains, as well as the experimental results of section 8, suggest that the rules have a fairly high degree of generality.

Putting all of these together, we have the fundamental result of the PECOS experiment: a demonstration that knowledge about a significant programming domain can be expressed effectively in a machine-usable form. The requirement of machine-usability is a very strong one. Most knowledge about programming is available only informally, couched in unstated assumptions. While such knowledge is understandable by people, it lacks the detail necessary for use by a machine. To a significant degree, PECOS's rules fill in some of the detail and the unstated assumptions. Taken together, the rules form a coherent body of knowledge that imposes a structure and taxonomy on part of the programming process; in effect, they constitute a step toward the development of a science of computer programming.

References

[Balzer, Goldman and Wile 1977]
Balzer, R., Goldman, N., and Wile, D. *Informality in program specifications.* **Proceedings of the Fifth International Joint Conference on Artificial Intelligence**, Cambridge, Massachusetts, August 1977, 389-397.

[Barstow and Kant 1976]
Barstow, D.R., and Kant, E. *Observations on the interaction of coding and efficiency knowledge in the PSI program synthesis system.* **Proceedings of the Second International Conference on Software Engineering**, San Francisco, California, October 1976, 19-31.

[Bobrow and Winograd 1977]
Bobrow, D. and Winograd, T. *An overview of KRL, a Knowledge representation language.* **Cognitive Science, 1**, January 1977, 3-46.

[Buchanan and Lederberg 1971]
Buchanan, B., and Lederberg, J. *The heuristic DENDRAL program for explaining empirical data.* IFIP, 1971, 179-188.

[Castaneda 1972]
Castaneda, Carlos. **Journey to Ixtlan: The Lessons of Don Juan**, Simon and Schuster, New York, 1972.

[Dahl, Dijkstra and Hoare 1972]
Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. **Structured Programming**, Academic Press, New York, 1972.

[Darlington and Burstall 1976]
Darlington, J. and Burstall, R.M. *A system which automatically improves programs.* **Acta Informatica, 6**, 1976, 41-60.

[Davis 1976]
Davis, R. *Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases.* Stanford University, Computer Science Department, AIM-283, July 1976.

[Davis and King 1977]
Davis, R., and King, J. *An overview of production systems*, in Elcock, E. W., and Michie, D. (Eds.) **Machine Representations of Knowledge**, Ellis Horwood Ltd. and John Wylie, 1977, 301-332.

[Davis, Buchanan and Shortliffe 1977]
Davis, R., Buchanan, B., and Shortliffe, E. *Production rules as a representation for a knowledge-based consultation program.* **Artificial Intelligence, 8**, February 1977, 15-45.

[Duda et al 1977]
Duda, R.O., Hart, P.E., Nilsson, N.J., and Sutherland, G.L. *Semantic network representation for rule-based inference systems.* Stanford Research Institute, Menlo Park, California, 1977.

[Ernst and Newell 1969]
Ernst, G.W. and Newell, A. **GPS: A Case Study in Generality and Problem Solving**, Academic Press, New York, 1969.

[Fikes and Nilsson 1971]
Fikes, R. and Nilsson, N. *STRIPS -- a new approach to the application of theorem proving to problem solving.* **Artificial Intelligence, 2,** Spring 1971.

[Ginsparg 1977]
Ginsparg, J. *A parser for English and its application in an automatic programming system.* forthcoming AI memo, Stanford University, 1977.

[Green 1969]
Green, C.C. *The application of theorem proving to question-answering systems.* Stanford University, Computer Science Department, AIM-96, August 1969.

[Green 1976]
Green, C.C. *The design of the PSI program synthesis system.* **Proceedings of the Second International Conference on Software Engineering,** San Francisco, California, October 1976, 4-18.

[Green 1977]
Green, C.C. *A summary of the PSI program synthesis system.* **Proceedings of the Fifth International Joint Conference on Artificial Intelligence,** Cambridge, Massachusetts, August 1977, 380-381.

[Green and Barstow 1975]
Green, C.C., and Barstow, D.R. *Some rules for the automatic synthesis of programs,* **Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,** Tbilisi, Georgia, USSR, September 1975, 232-239.

[Green and Barstow 1977a]
Green, C.C., and Barstow, D.R. *A hypothetical dialogue exhibiting a knowledge base for a program understanding system,* in Elcock, E. W., and Michie, D. (Eds.) **Machine Representations of Knowledge,** Ellis Horwood Ltd. and John Wylie, 1977, 335-359.

[Green and Barstow 1977b]
Green, C.C., and Barstow, D.R. *Program synthesis knowledge for efficient sorting.* (in preparation).

[Green et al 1974]
Green, C.C., Waldinger, R.J., Barstow, D.R., Elschlager, R., Lenat, D.B., McCune, B.P., Shaw, D.E., and Steinberg, L.I., *Progress report on program understanding systems.* Stanford University, Computer Science Department, AIM-240, August 1974.

[Gries 1977]
Gries, D. *A linear sieve algorithm for finding prime numbers.* Cornell University, Department of Computer Science, TR 77-313, 1977.

[Hewitt 1972]
Hewitt, C. *Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot.* Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI-TR-258, April 1972.

[Kant 1977]
Kant, E. *The selection of efficient implementations for a high level language.* Proceedings of ACM SIGART-SIGPLAN **Symposium on Artificial Intelligence and Programming Languages**, August 1977, 140-146.

[Knuth 1973]
Knuth, D.E. **The Art of Computer Programming, Sorting and Searching** (vol. 3). Addison-Wesley, Menlo Park, California, 1973.

[Knuth 1977]
Knuth, D.E. **The Art of Computer Programming, Combinatorial Algorithms** (vol. 4). Addison-Wesley, 1977 (preprint).

[Lenat 1976]
Lenat, D.B. *AM: an artificial intelligence approach to discovery in mathematics as heuristic search.* Stanford University, Computer Science Department, AIM-286, July 1976.

[Lenat and Harris 1977]
Lenat, D.B., and Harris, G. *Designing a rule system that searches for scientific discoveries.* to appear in Waterman, D.A. and Hayes-Roth, F. (Eds.) **Pattern-Directed Inference Systems**, Academic Press, 1977.

[Liskov et al 1977]
Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. *Abstraction mechanisms in CLU.* **Communications of the ACM**, 20, August 1977, 564-576.

[Low 1974]
Low, J. *Automatic coding: choice of data structures.* Stanford University, Computer Science Department, AIM-242, August 1974.

[Ludlow 1977]
Ludlow, J. Masters Project. Stanford University, 1977.

[Macsyma 1974]
The MACSYMA reference manual, The MATHLAB Group, Massachusetts Institute of Technology, September 1974.

[Mairson 1977]
Mairson, H.G. *Some new upper bounds on the generation of prime numbers.* **Communications of the ACM, 20**, September 1977, 664-669.

[Manna and Waldinger 1977]
Manna, Z., and Waldinger, R. *The automatic synthesis of recursive programs.* Proceedings of ACM SIGART-SIGPLAN **Symposium on Artificial Intelligence and Programming Languages**, August 1977, 29-36.

[MCAP 1954]
Symposium on Automatic Programming for Digital Computers. Mathematical Computing Advisory Panel, Office of Naval Research, Department of the Navy, Washington, D.C., May 1954.

[McCune 1977]
McCune, B.P. *The PSI program model builder: synthesis of very high-level programs.* Proceedings of ACM SIGART-SIGPLAN **Symposium on Artificial Intelligence and Programming Languages**, August 1977, 130-139.

[Phillips 1977]
Phillips, J. *Program inference from traces using multiple knowledge sources.* **Proceedings of the Fifth International Joint Conference on Artificial Intelligence**, Cambridge, Massachusetts, August 1977, 812.

[Reiser 1976]
Reiser, J.F. (Ed.) *SAIL Reference Manual.* Stanford University, Computer Science Department, AIM-289, August 1976.

[Rich and Shrobe 1976]
Rich, C. and Shrobe, H. *Initial report on a LISP programmer's apprentice.* Massachusetts Institute of Technology, AI-TR-354, December 1976.

[Robinson and Levitt 1977]
Robinson, L. and Levitt K.N. *Proof techniques for hierarchically structured programs.* **Communications of the ACM, 20**, April 1977, 271-283.

[Rovner 1976]
Rovner, P.D. *Automatic Representation Selection for Associative Data Structures.* The University of Rochester, Computer Science Department, TR10, September 1976.

[Ruth 1976]
Ruth, G. *Intelligent program analysis.* **Artificial Intelligence, 7,** Spring 1976, 65-85.

[Sacerdoti 1975]
Sacerdoti, E.D. *The nonlinear nature of plans.* **Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,** Tbilisi, Georgia, USSR, September 1975, 206-214.

[Schwartz 1975]
Schwartz, J.T. *On programming: an interim report on the SETL project.* New York University, Courant Institute of Mathematical Sciences, Computer Science Department, June 1975.

[Shortliffe 1974]
Shortliffe, E.H. **MYCIN: Computer-Based Medical Consultations,** American Elsevier, New York, 1976.

[Standish et al 1976]
Standish, T., Harriman, D., Kibler, D., and Neighbors, J. *The Irvine program transformation catalogue.* University of California at Irvine, Computer Science Department, January 1976.

[Sussman 1975]
Sussman, G.J. **A Computer Model of Skill Acquisition,** American Elsevier, New York, 1975.

[Teitelman 1975]
Teitelman, W. **INTERLISP Reference Manual.** Xerox Palo Alto Research Center, Palo Alto, California, December 1975.

[Thorelli 1972]
Thorelli, L.-E. *Marking algorithms.* **Behandling Informations-tidskrift for Nordisk, 12,** 1972, pages 555-568.

[von Henke and Luckham 1974]
v. Henke, F.W. and Luckham, D.C. *Automatic program verification III: a methodology for verifying programs.* Stanford University, Computer Science Department, AIM-256, December 1974.

[Wilber 1976]
Wilber, B.M. **A QLISP Reference Manual.** Stanford Research Institute, Menlo Park, California, 1976.

[Winston 1975]
Winston, P.H. *Learning Structural Descriptions from Examples.* in Winston, P.H. (Ed.) **The Psychology of Computer Vision,** McGraw-Hill, 1975.

[Wirth 1971]
Wirth, N. *Program development by stepwise refinement.* **Communications of the ACM**, 14, April 1971, 221-227.

[Wulf, London and Shaw 1976]
Wulf W., London, R., and Shaw, M. *An introduction to the construction and verification of ALPHARD programs.* **IEEE Transactions on Software Engineering**, December 1976, pages 253-265.

[Zahn 1974]
Zahn, C.T. *A control statement for natural top-down structured programming*, **Symposium on Programming Languages**, Paris, 1974.

Appendix 1. CONDITIONS AND ACTIONS EXPRESSIBLE IN RULES

This appendix contains a complete list of the pattern types that may occur in PECOS's rules. Patterns are matched in the context of a "current expression." The current expression is held as the value of the special variable #. Many of the pattern types rebind # before attempting any subpatterns; # will be used to stand for this value, except that in expressions like "rebinds # ..." it will stand for the variable itself. Hopefully, there will be no confusion and the resulting pattern descriptions somewhat easier to follow. Other notational conventions that will be used are:

[x] will denote the result of evaluating x
x::y will denote the value of the y property of x

Included in the description of each pattern type are specifications of any variables that are bound by such patterns, any expressions that must be checked for their use of free variables, whether or not such a pattern could fail directly, and whether any subtasks might be generated by such a pattern.

The patterns are divided into three categories, those that occur in rule conditions, those that occur in rule actions, and several rather esoteric patterns that are conceptually related although some appear in conditions and some in actions.

1.1. Pattern types in conditions

1.1.1. Conditions on nodes

All of these condition patterns fail if the current expression (the value of #) is not a node.

(#C pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none

Rebinds # to the concept of # and attempts to match the pat$_i$.

(?CONCEPT-CLASS class)
Bound variables: none
Free expressions: none
May fail at this level

Succeeds if # is a node whose concept has as its class class.

(#REF concept pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none
Possible subtask: **(REFINE <most refined node for #>)**
May fail at this level

Succeeds if #, or any refinement of #, matches the **concept** and all of the **pat$_1$**. If none of the refinements of # match, but there is a sequence of rules that refines the most refined node under # into a node that matches the **concept**, the subtask is generated. Otherwise the match fails. **concept** must be either a **<concept name>** or an expression of the form **(?CONCEPT-CLASS class)**, where **class** is the name of a concept class.

1.1.2. Conditions on node properties

All of these condition patterns fail if # is not a node. They all involve various ways of checking conditions of properties of nodes.

(#P pname pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none
Possible subtask: **(PROPERTY pname #)**

Rebinds # to **#::pname** and checks the **pat$_i$**. The subtask is generated if the property does not exist yet.

(#P* expr pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: expr
Possible subtask: **(PROPERTY [expr] #)**

The same as **#P** except that **expr** is evaluated to determine the name of the property.

(#P/NIL pname pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none

The same as #P except that if the property does not exist for #, a value of NIL is assumed, and no subtask is generated.

(#P/NOTNIL pname pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none
Possible subtask: **(PROPERTY pname #)**

The same as **#P** except that the match fails if the property's value is NIL.

(#P: var pat$_1$ pat$_2$ ...)

Parameter variables: **var**
Bound variables: none
Free expressions: none

Rebinds # to the value of any property of #, and attempts to match the **pat$_i$**. If the match succeeds, returns a binding of **var** to the name of the property. This is one of the pattern types that can succeed in several ways. Hence, **var** is considered to be a parameter. (Before such an expression is added to an applicability or binding pattern, the **#P:** is replaced by **#P*** so that such expressions may evaluate **var** to determine the property being examined. This is, of course, invisible to the rule writer. As far he or she is concerned, **#P:** patterns are simply patterns that can succeed in several ways.)

(#RDS pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none
Possible subtask: **(PROPERTY RESULT-DATA-STRUCTURE #)**

Rebinds # to #::RESULT-DATA-STRUCTURE, and evaluates the pat$_i$. If the property does not exist yet, the subtask is generated.

(#DS pname pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none
Possible subtask: **(PROPERTY pname #)**
Possible subtask: **(PROPERTY RESULT-DATA-STRUCTURE #::pname)**

Rebinds # to (#::pname)::RESULT-DATA-STRUCTURE, and attempts the pat$_i$. If either property does not exist yet, the appropriate subtask is generated.

1.1.3. Queries

Queries provide a way of testing more complex conditions than are conveniently expressible in other forms of patterns. Their principal value is due to the fact that the same query may be answered by any of several rules. Thus, they permit the use of rules to test conditions. There is currently only one query pattern.

(?QUERY rel arg_1 arg_2 ...)
    Bound variables: #ANSWER
    Free expressions: arg_i
    May fail at this level
    Possible subtask: (QUERY rel [arg_1] [arg_2] ...)

The arg_i are evaluated to determine the query shown above. If this query has already been answered, the pattern succeeds and returns a binding of #ANSWER to the query's answer. Otherwise the subtask is generated.

1.1.4. Conditions on various structures

As property values may be structures other than nodes, there are pattern types that test conditions on other structures.

(?#= expr)
    Bound variables: none
    Free expressions: none
    May fail at this level

Succeeds if # is equal to expr.

(?#=* expr)
    Bound variables: none
    Free expressions: expr
    May fail at this level

The same as ?#= except that expr is evaluated.

(?#=/NOT expr)
Bound variables: none
Free expressions: none
May fail at this level

Succeeds if # is not equal to expr.

1.1.5. Conditions on lists

Frequently, a property value will be a list. The elements of such a list are often nodes but they may be arbitrary structures. Several pattern types are available for testing conditions on such lists.

(#ALL← var expr)
Bound variables: **var**
Free expressions: **expr**

For every element of #, rebinds # to that element and evaluates expr. Returns a binding of var to the list formed by collecting the results of these evaluations.

(#ALL← var NIL pat$_1$ pat$_2$ ...)
Bound variables: **var$_1$**
Free expressions: none
May fail at this level

For every element of #, rebinds # to that element and attempts the pat$_i$. If all succeed then returns a binding of var to a top level copy of #.

(#ALL← var$_1$ var$_2$ pat$_1$ pat$_2$ ...)
Bound variables: **var$_1$**
Free expressions: none
May fail at this level

For every element of #, rebinds # to that element and attempts the pat$_i$. If all succeed then returns a binding of var$_1$ to the list formed by collecting the binding of var$_2$ determined while attempting the pat$_i$.

(#SUBSET← var expr)
Bound variables: var
Free expressions: expr

For every element of #, rebinds # to that element and evaluates expr. Returns a binding of var to the list formed by collecting the non-NIL results of these evaluations.

(#SUBSET← var NIL pat_1 pat_2 ...)
Bound variables: var_1
Free expressions: none

For every element of #, rebinds # to that element and attempts the pat_i. Returns a binding of var to a list of those elements for which the attempt succeeded.

(#SUBSET← var_1 var_2 pat_1 pat_2 ...)
Bound variables: var_1
Free expressions: none

For every element of #, rebinds # to that element and attempts the pat_i. Returns a binding of var_1 to the list formed by collecting the binding of var_2 determined while attempting the pat_i for all values of # for which the attempt succeeded.

(?#ALL pat_1 pat_2 ...)
Bound variables: none
Free expressions: none
May fail at this level

For every element of #, rebinds # to that element and attempts the pat_i. If all succeed then the ?#ALL pattern succeeds.

(?#NONE pat_1 pat_2 ...)
Bound variables: none
Free expressions: none
May fail at this level

For every element of #, rebinds # to that element and attempts the pat_i. If none succeed then the ?#ALL pattern succeeds.

1.1.6. Patterns on segments of lists

The patterns of the previous section dealt with individual elements of lists. In addition, there are pattern types for dealing with segments of lists. As a list has many ways to be broken into segments, such patterns may succeed in several ways. Hence, they are considered to be parameterized patterns, but this fact is again invisible to the user and rule writer.

(?#: pat$_1$ pat$_2$ ...)

Bound variables: none
Free expressions: none
May fail at this level

Each of the **pat$_i$** must be an **EL:** or a **SEG:** pattern (see below). Attempts to find all possible matches of the **pat$_i$** with the value of **#**.

(#EL: var pat$_1$ pat$_2$ ...)

Parameter variables: **var**
Bound variables: none
Free expressions: none
May fail at this level

Rebinds **#** to the first element of **#** and attempts the **pat$_i$**. **var** is considered to be a parameter for consistency with **#SEG:**.

(#SEG: var pat$_1$ pat$_2$ ...)

Parameter variables: **var**
Bound variables: none
Free expressions: none

Rebinds **#** to any segment (including the empty segment) of **#** and attempts the **pat$_i$**. **var** is considered to be a parameter since many segments may succeed.

1.1.7. Patterns that simply provide bindings

Several patterns always succeed and only return bindings. Note that several of the patterns described above also return bindings in addition to their other functions.

(←← var)
Bound variables: var
Free expressions: none

Binds var to the current expression (#).

(←← var expr)
Bound variables: var
Free expressions: expr

Binds var to [expr].

(←← var_1 var_2 pat_1 pat_2 ...)
Bound variables: var
Free expressions: none

Binds var_1 to the binding found for var_2 after matching all of the pat_i.

1.1.8. Other pattern types

There are several other pattern types that do not fit easily into the above categorization.

(#← expr pat_1 pat_2 ...)
Bound variables: none
Free expressions: expr

Rebinds # to [expr] and checks the pat_i. This provides a way of specifying arbitrary values to be matched against.

(#NOT pat_1 pat_2 ...)
Bound variables: none
Free expressions: none

Succeeds if the any of the pat_i fail to match the current expression.

1.2. Pattern types that appear in rule actions

1.2.1. Patterns that return pointers to nodes

All of these patterns return pointers to nodes. In addition, they rebind # to the node before evaluating any subpatterns. Such subpatterns are usually used to attach properties to the nodes.

(#NEW concept pat_1 pat_2 ...)
 Bound variables: none
 Free expressions: none

Creates a new node with concept **concept**, rebinds # to that node, and evaluates the pat_i.

(#SAME expr pat_1 pat_2 ...)
 Bound variables: none
 Free expressions: expr

Rebinds # to [expr] and evaluates the pat_i. This is normally used when attaching an old node as a refinement of some other node.

1.2.2. Patterns that attach properties to nodes

(←#P pname expr)
 Bound variables: none
 Free expressions: expr

Attaches the value of **expr** as the **pname** property to # (which must be a node).

(←#P* pname expr)
 Bound variables: none
 Free expressions: none

The same as **←#P** except that **expr** is not evaluated.

(←#P/NOTNIL pname expr)
Bound variables: none
Free expressions: expr

The same as ←#P except that it is only done if [expr] is not NIL.

(←#RDS expr)
Bound variables: none
Free expressions: expr

Attaches [expr] as the RESULT-DATA-STRUCTURE property of #.

1.2.3. Patterns that perform other kinds of computations

These were included in order to avoid the use of "arbitrary LISP expressions" in the rules. (But I hereby confess to having included just such arbitrary expressions at various times in rules that I have written.)

(#LOCAL← var)
Bound variables: var
Free expressions: none

Binds var to #.

(#LOCAL← var expr)
Bound variables: var
Free expressions: expr

Binds var to the result of evaluating expr.

(#IF $expr_1$ $expr_2$ $expr_3$)
Bound variables: none
Free expressions: $expr_1$, $expr_2$, $expr_3$

$expr_1$ is evaluated. If the result is non-NIL, then $expr_2$ is evaluated. Otherwise $expr_3$ is evaluated.

(#JOIN var expr_1 expr_2)
Bound variables: **var**
Free expressions: **expr_1, expr_2**

expr_1 is evaluated and the result should be a list. This list is mapped down, binding **var** to each element. The results of evaluating **expr_2** for each value are joined together and returned.

(#COLLECT var expr_1 expr_2)
Bound variables: **var**
Free expressions: **expr_1, expr_2**

expr_1 is evaluated and the result should be a list. This list is mapped down, binding **var** to each element. The results of evaluating **expr_2** for each value are collected into a single list which is returned.

1.2.4. Patterns that perform refinements

Note that this does not appear in any rule statement, but is rather implied by a rule being a REF← rule.

(←#REF expr)
Bound variables: none
Free expressions: **expr**

Attaches **[expr]** as a refinement of **#**.

1.3. Esoteric patterns

1.3.1. Patterns involving pairs of values

Frequently it has been found necessary or useful to have a way of associating two values, usually in a list of such associations, where the list is the value of some property of some node. While it would have been possible to use such LISP functions as CONS, CAR, CDR, and ASSOC to deal with such situations, a bias against allowing arbitrary LISP code in rules has resulted in a set of patterns that can put together and take apart such pairs. The objects dealt with are termed **<pair>**s and each has two parts, termed the first part and the second part.

(#PAIR name expr)
    Bound variables: none
    Free expressions: expr

Creates a <pair> from name and [expr].

(#PAIR* $expr_1$ $expr_2$)
    Bound variables: none
    Free expressions: $expr_1$, $expr_2$

Creates a <pair> from [$expr_1$] and [$expr_2$].

(#FIRST/VALUE expr)
    Bound variables: none
    Free expressions: expr

expr is evaluated and result must be a <pair>. Returns the first part of [expr].

(#SECOND/VALUE expr)
    Bound variables: none
    Free expressions: expr

expr is evaluated and result must be a <pair>. Returns the first part of [expr].

(#PAIRED*/VALUE $expr_1$ $expr_2$)
    Bound variables: none
    Free expressions: $expr_1$, $expr_2$

$expr_2$ is evaluated and the result must be a list of <pair>s. Returns the second part of first element of [$expr_2$] whose first part is [$expr_1$].

(#PAIRED name pat_1 pat_2 ...)
    Bound variables: none
    Free expressions: none

must be a list of <pair>s. Rebinds # to that <pair> whose first part is name and attempts the pat_i.

(#PAIRED* expr pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: **expr**

must be a list of <pair>s. Rebinds **#** to that <pair> whose first part is **[expr]** and attempts the **pat**$_i$.

(#FIRST pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none

must be a <pair>. Rebinds **#** to the first part of **#** and attempts the **pat**$_i$.

(#SECOND pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none

must be a <pair>. Rebinds **#** to the second part of **#** and attempts the **pat**$_i$.

1.3.2. Global associations

The use of <pair>s permits the use of local associations (associations within a localized list of such associations). In many cases, it has been necessary to have ways of making global associations. For example, it is useful to be able to refer to a particular memory structure through such a global association on the label of it memory unit. The patterns given below provide this facility.

(#GLOBAL← var)
Bound variables: **var**
Free expressions: none

Sets up a global association identifier for **#**, and binds **var** to that identifier.

(#GLOBAL pat$_1$ pat$_2$ ...)
Bound variables: none
Free expressions: none

must be a global association identifier. Rebinds **#** to the associated value and attempts the **pat**$_i$.

1.3.3. Pattern for the use of the efficiency expert

There are several pattern types that have been included for the use the efficiency expert. They are included here for completeness.

(#P/EFF ...)

There is a separate category of properties for the exclusive use of the efficiency expert. This pattern is just like #P except that it deals with the efficiency properties.

(←#P/EFF ...)

Just like ←#P except that it deals with efficiency properties.

(#GLOBAL←/EFF ...)

There is also a global association list available to the efficiency expert. This pattern is the same as #GLOBAL← except that it deals with this special association list.

(#GLOBAL/EFF ...)

Similarly, this deals with the efficiency expert's association list.

1.3.4. Patterns for use by other aspects of the system

There are also two other categories of patterns for use by the system. STATE properties are intended for use in keeping track of the state of the computation. SYS properties are intended for use in maintaining various back pointers required by the system. Currently no rules use either of these kinds of properties.

(#P/STATE ...)

Same as #P but for the state properties.

(←#P/STATE ...)

Same as ←#P but for the state properties.

(#P/SYS ...)

Same as #P but for the "system" properties.

(←#P/SYS ...)

Same as ←#P but for the "system" properties.